MIPS32™ 4KE™ Processor Cores Software User's Manual

Document Number: MD00103
Revision 02.00
November 8, 2002

MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043-1353

Template: B1.07, Build with Conditional Tags: 2B EMERALD MIPS32 PROC

Table of Contents

List of Figures

List of Tables

Introduction to the MIPS32™ 4KE™ Processor Core Family

The MIPS32™ 4KE™ core from MIPS® Technologies is a high-performance, low-power, 32-bit MIPS RISC processor core family intended for custom system-on-silicon applications. The core is designed for semiconductor manufacturing companies, ASIC developers, and system OEMs who want to rapidly integrate their own custom logic and peripherals with a high-performance RISC processor. A 4KE core is fully synthesizable to allow maximum flexibility; it is highly portable across processes and can easily be integrated into full system-on-silicon designs. This allows developers to focus their attention on end-user specific characteristics of their product.

The 4KE core is ideally positioned to support new products for emerging segments of the digital consumer, network, systems, and information management markets, enabling new tailored solutions for embedded applications.

The 4KE family has three members: the 4KEc™, 4KEm™, and 4KEp™ cores. The three devices differ mainly in the type of multiply-divide unit (MDU) and the Memory Management Unit (MMU).

- The 4KEc core contains a fully-associative Translation Lookaside Buffer (TLB)-based MMU and pipelined MDU.
- The 4KEm core contains a fixed mapping translation (FMT) mechanism in the MMU, that is smaller and simpler than the TLB-based implementation used in the 4KEc core. A pipelined MDU (like the 4KEc core) is used.
- The 4KEp core contains the same FMT-based MMU (like the 4KEm core), but a smaller non-pipelined MDU.

The term *4KE core* as used in this document, generally refers to all cores in the 4KE family. When referring to characteristics unique to an individual family member, the specific core type (*4KEc*, *4KEm*, or *4KEp core*) will be identified.

On a 4KE core, instruction and data caches are optional and fully programmable from 0 - 64 Kbytes in size. In addition, each cache can be organized as direct-mapped, 2-way, 3-way, or 4-way set associative. On a cache miss, loads are blocked only until the first critical word becomes available. The pipeline resumes execution while the remaining words are being written to the cache. Both caches are virtually indexed and physically tagged. Virtual indexing allows the cache to be indexed in the same clock in which the address is generated rather than waiting for the virtual-to-physical address translation in the TLB.

The core implements the MIPS32 Release 2 Instruction Set Architecture (ISA), and may optionally support the MIPS16e Application Specific Extension (ASE) for code compression. The MMU of the 4KEc core contains a 4-entry instruction TLB (ITLB), a 4-entry data TLB(DTLB), and a 16 dual-entry joint TLB (JTLB) with variable page sizes. The 4KEm and 4KEp cores contain a simplified fixed mapping translation (FMT) mechanism, for applications that do not require the full capabilities of a TLB.

The 4KEc and 4KEm Multiply-Divide Unit (MDU) supports a maximum issue rate of one 32x16 multiply (MUL/MULT/MULTU), multiply-add (MADD/MADDU), or multiply-subtract (MSUB/MSUBU) operation per clock, or one 32x32 MUL, MADD, or MSUB every other clock. The MDU on the 4KEp core uses an area-sensitive iterative algorithm.

The basic Enhanced JTAG (EJTAG) features provide CPU run control with stop, single stepping and re-start, and with software breakpoints through the SDBBP instruction. Additional EJTAG features - instruction and data virtual address hardware breakpoints, connection to an external EJTAG probe through the Test Access Port (TAP), and PC/Data tracing, may optionally be included.

.The rest of this chapter provides an overview of the MIPS32 4KE processor core and consists of the following sections:

- Section 1.1, "Features"

- Section 1.2, "4KE Block Diagram"

1.1 Features

- 5-stage pipeline
- 32-bit Address and Data Paths
- MIPS32-Compatible Instruction Set
  - Multiply-add and multiply-subtract instructions (MADD, MADDU, MSUB, MSUBU)
  - Targeted multiply instruction (MUL)
  - Zero and one detect instructions (CLZ, CLO)
  - Wait instruction (WAIT)
  - Conditional move instructions (MOVZ, MOVN)
  - Prefetch instruction (PREF)
- MIPS32 Enhanced Architecture (Release 2) Features
  - Vectored interrupts and support for an external interrupt controller
  - Programmable exception vector base
  - Atomic interrupt enable/disable
  - GPR shadow sets
  - Bit field manipulation instructions
  - Improved virtual memory support (smaller page sizes and hooks for more extensive page table manipulation)
- MIPS16e Application Specific Extension
  - 16 bit encodings of 32-bit instructions to improve code density
  - Special PC-relative instructions for efficient loading of addresses and constants
  - Data type conversion instructions (ZEB, SEB, ZEH, SEH)
  - Compact jumps (JRC, JALRC)
  - Stack frame set-up and tear down “macro” instructions (SAVE and RESTORE)
- Programmable Cache Sizes
  - Individually configurable instruction and data caches
  - Sizes from 0 up to 64 Kbytes
  - Direct-mapped, or 2-, 3-, 4-Way set associative
  - Loads that miss in the cache are blocked only until critical word is available
  - Supports Write-back with write-allocation and Write-through with or without write-allocation
  - 128-bit (16-byte) cache line size, word sectored - suitable for standard 32-bit wide single-port SRAM
  - Virtually indexed, physically tagged
  - Cache line locking support
  - Non-blocking prefetches

- Scratchpad RAM support
  - Replace one way of instruction cache and/or data cache
  - Maximum 20-bit index (1M address)
  - Memory-mapped registers attached to scratchpad port can be used as a coprocessor interface
- R4000 Style Privileged Resource Architecture
  - Count/compare registers for real-time timer interrupts
  - Instruction and data watch registers for software breakpoints
- Programmable Memory Management Unit (4KEc core only)
  - 16 dual-entry MIPS32-style JTLB with variable page sizes
  - 4-entry instruction TLB
  - 4-entry data TLB
- Programmable Memory Management Unit (4KEm and 4KEp cores only)
  - Simple Fixed Mapping Translation (FMT)
  - Address spaces mapped using register bits
- Simple Bus Interface Unit (BIU)
  - All I/Os fully registered
  - Separate unidirectional 32-bit address and data buses
  - Two 16-byte collapsing write buffers
- CorExtend™ User Defined Instruction capability (access to this feature is available in the 4KE Pro™ cores and requires a separate license)
  - Optional support for the CorExtend feature allows users to define and add instructions to the core (as a build-time option)
  - Single or multi-cycle instructions
  - Source operations from register, immediate field, or local state
  - Destination to a register or local state
- Full featured Coprocessor 2 Interface
  - Almost all I/Os registered
  - Separate unidirectional 32-bit instruction and data buses
  - Support for branch on Coprocessor condition
  - Processor to/from Coprocessor register data transfers
  - Direct memory to/from Coprocessor register data transfers
- Multiply-Divide Unit (4KEc and 4KEm cores)
  - Maximum issue rate of one 32x16 multiply per clock
  - Maximum issue rate of one 32x32 multiply every other clock
  - Early-in divide control. Minimum 11, maximum 34 clock latency on divide
- Multiply-Divide Unit (4KEp core)
  - Iterative multiply and divide. 32 or more cycles for each instruction.

- Power Control
  - No minimum frequency
  - Power-down mode (triggered by WAIT instruction)
  - Support for software-controlled clock divider
  - Support for extensive use of fine-grain clock gating
- EJTAG Debug Support
  - CPU control with start, stop and single stepping
  - Software breakpoints via the SDBBP instruction
  - Optional hardware breakpoints on virtual addresses; 4 instruction and 2 data breakpoints, 2 instruction and 1 data breakpoint, or no breakpoints
  - Optional Test Access Port (TAP) facilitates high speed download of application code
  - Optional EJTAG Trace hardware to enable real-time tracing of executed code

1.2 4KE Block Diagram

The 4KE core contains both required and optional blocks, as shown in the block diagram in Figure 1-1 on page 5. Required blocks are the lightly shaded areas of the block diagram and are always present in any core implementation. Optional blocks may be added to the base core, depending on the needs of a specific implementation. The required blocks are as follows:

- Execution Unit
- Multiply-Divide Unit (MDU)
- System Control Coprocessor (CP0)
- Memory Management Unit (MMU)
- Cache Controller
- Bus Interface Unit (BIU)
- Power Management

Optional blocks include:

- Instruction Cache (I-cache)
- Data Cache (D-cache)
- Enhanced JTAG (EJTAG) Controller
- MIPS16e support
- Coprocessor 2 Interface (CP2)
- CorExtend™ User Defined Instructions (UDI)

Figure 1-1 shows a block diagram of a 4KE core. The MMU can be implemented using either a translation lookaside buffer in the case of the 4KEc core, or a fixed mapping (FMT) in the case of the 4KEm and 4KEp cores. Refer to Chapter 3, “Memory Management,” on page 33 for more information.

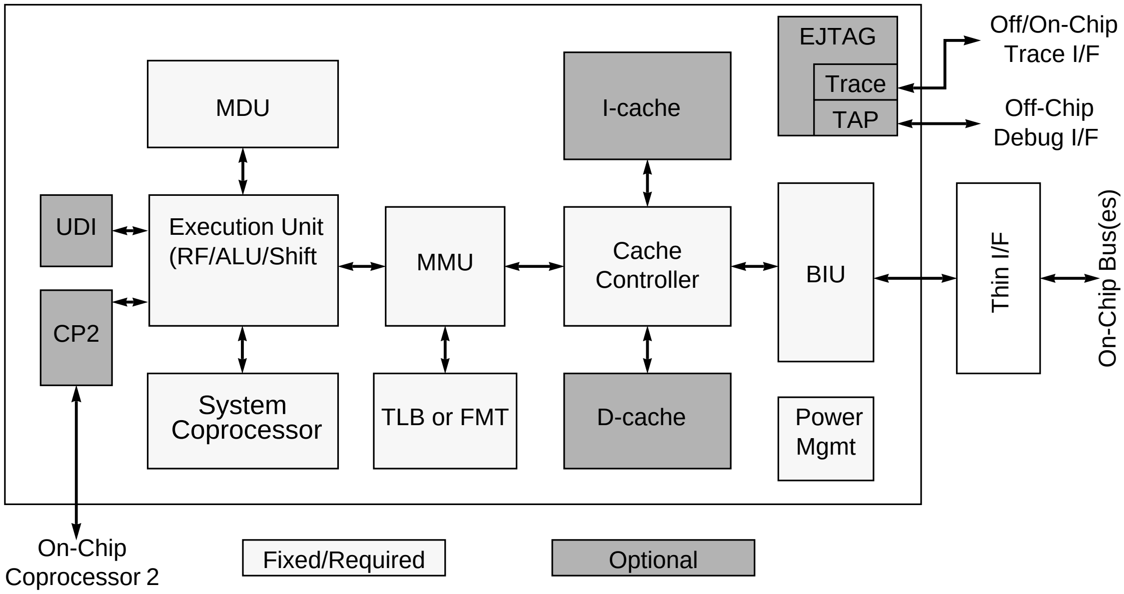


Figure 1-1 4KE Processor Core Block Diagram

1.2.1 Required Logic Blocks

The following subsections describe the various required logic blocks of the 4KE processor core.

1.2.1.1 Execution Unit

The core execution unit implements a load-store architecture with single-cycle Arithmetic Logic Unit (ALU) operations (logical, shift, add, subtract) and an autonomous multiply-divide unit. The core contains thirty-two 32-bit general-purpose registers(GPRs) used for scalar integer operations and address calculation. Optionally, one or three additional register file shadow sets (each containing thirty-two registers) can be added to minimize context switching overhead during interrupt/exception processing. The register file consists of two read ports and one write port and is fully bypassed to minimize operation latency in the pipeline.

The execution unit includes:

- 32-bit adder used for calculating the data address
- Address unit for calculating the next instruction address
- Logic for branch determination and branch target address calculation
- Load aligner
- Bypass multiplexers used to avoid stalls when executing instruction streams where data-producing instructions are followed closely by consumers of their results
- Zero/One detect unit for implementing the CLZ and CLO instructions
- ALU for performing bitwise logical operations
- Shifter and Store aligner

1.2.1.2 Multiply/Divide Unit (MDU)

The Multiply/Divide unit performs multiply and divide operations. In the 4KEc and 4KEm processors, the MDU consists of a 32x16 booth-encoded multiplier, result-accumulation registers (HI and LO), multiply and divide state machines, and

all multiplexers and control logic required to perform these functions. This pipelined MDU supports execution of a 16x16 or 32x16 multiply operation every clock cycle; 32x32 multiply operations can be issued every other clock cycle. Appropriate interlocks are implemented to stall the issue of back-to-back 32x32 multiply operations. Divide operations are implemented with a simple 1 bit per clock iterative algorithm and require 35 clock cycles in worst case to complete. Early-in to the algorithm detects sign extension of the dividend, if it is actual size is 24, 16 or 8 bit. the divider will skip 7, 15 or 23 of the 32 iterations. An attempt to issue a subsequent MDU instruction while a divide is still active causes a pipeline stall until the divide operation is completed.

The area-efficient, non-pipelined MDU in the 4KEp core consists of a 32-bit full-adder, result-accumulation registers (HI and LO), a combined multiply/divide state machine, and all multiplexers and control logic required to perform these functions. It performs any multiply using 32 cycles in an iterative 1 bit per clock algorithm. Divide operations are also implemented with a simple 1 bit per clock iterative algorithm (no early-in) and require 35 clock cycles to complete. An attempt to issue a subsequent MDU instruction while a multiply/divide is still active causes a pipeline stall until the operation is completed.

The 4KE implements an additional multiply instruction, MUL, which specifies that lower 32-bits of the multiply result be placed in the register file instead of the HI/LO register pair. By avoiding the explicit move from LO (MFLO) instruction, required when using the LO register, and by supporting multiple destination registers, the throughput of multiply-intensive operations is increased.

Two instructions, multiply-add (MADD/MADDU) and multiply-subtract (MSUB/MSUBU), are used to perform the multiply-add and multiply-subtract operations. The MADD instruction multiplies two numbers and then adds the product to the current contents of the HI and LO registers. Similarly, the MSUB instruction multiplies two operands and then subtracts the product from the HI and LO registers. The MADD/MADDU and MSUB/MSUBU operations are commonly used in Digital Signal Processor (DSP) algorithms.

1.2.1.3 System Control Coprocessor (CP0)

In the MIPS architecture, CP0 is responsible for the virtual-to-physical address translation, cache protocols, the exception control system, the processor's diagnostics capability, operating mode selection (kernel vs. user mode), and the enabling/disabling of interrupts. Configuration information such as cache size, set associativity, and presence of build-time options are available by accessing the CP0 registers. Refer to Chapter 5, “CP0 Registers,” on page 87 for more information on the CP0 registers. Refer to Chapter 9, “EJTAG Debug Support,” on page 165 for more information on EJTAG debug registers.

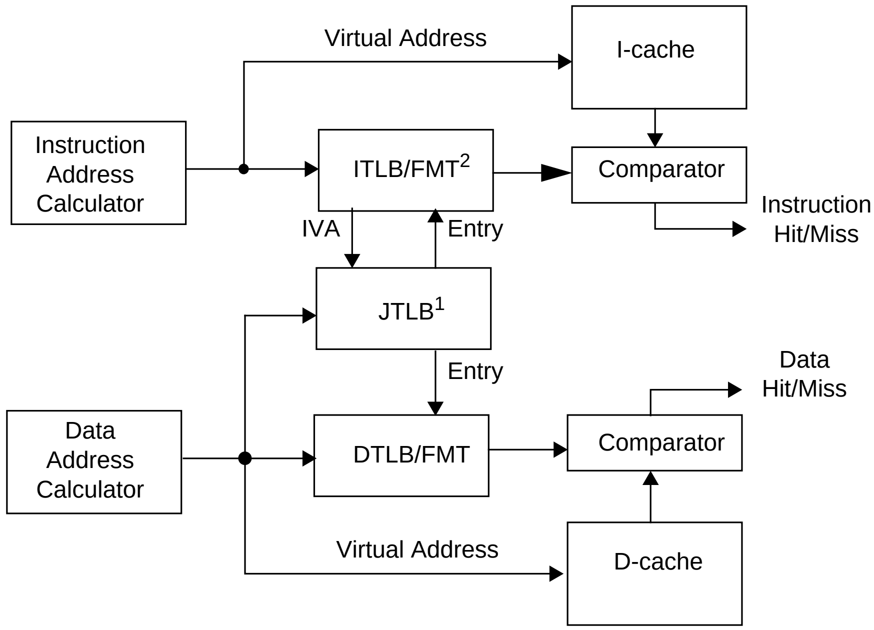
1.2.1.4 Memory Management Unit (MMU)

The 4KE core contains an MMU that interfaces between the execution unit and the cache controller, shown in Figure 1-2 on page 7. Although the 4KEc core implements a 32-bit architecture, the Memory Management Unit (MMU) is modeled after the MMU found in the 64-bit R4000 family, as defined by the MIPS32 architecture.

The 4KEc core implements its MMU based on a Translation Lookaside Buffer (TLB). The TLB consists of three translation buffers: a 16 dual-entry fully associative Joint TLB (JTLB), a 4-entry fully associative Instruction TLB (ITLB) and a 4-entry fully associative data TLB (DTLB). The ITLB and DTLB, also referred to as the micro TLBs, are managed by the hardware and are not software visible. The micro TLBs contain subsets of the JTLB. When translating addresses, the corresponding micro TLB (I or D) is accessed first. If there is not a matching entry, the JTLB is used to translate the address and refill the micro TLB. If the entry is not found in the JTLB, then an exception is taken. To minimize the micro TLB miss penalty, the JTLB is looked up in parallel with the DTLB for data references. This results in a one cycle stall for a DTLB miss and a two cycle stall for an ITLB miss.

The 4KEm and 4KEp cores implement a FMT-based MMU instead of a TLB-based MMU. The FMT replaces the JTLB, ITLB and DTLB in the 4KEc core. The FMT performs a simple translation to get the physical address from the virtual address. Refer to Chapter 3, “Memory Management,” on page 33 for more information on the FMT.

Figure 1-2 on page 7 shows how the address translation mechanism interacts with cache access. The JTLB in this figure is only present on the 4KEc core.



1. JTLB only exists in the 4KEc core
2. ITLB/DTLB implemented in the 4KEc core only. FMT implemented inthe 4KEm and 4KEp cores.

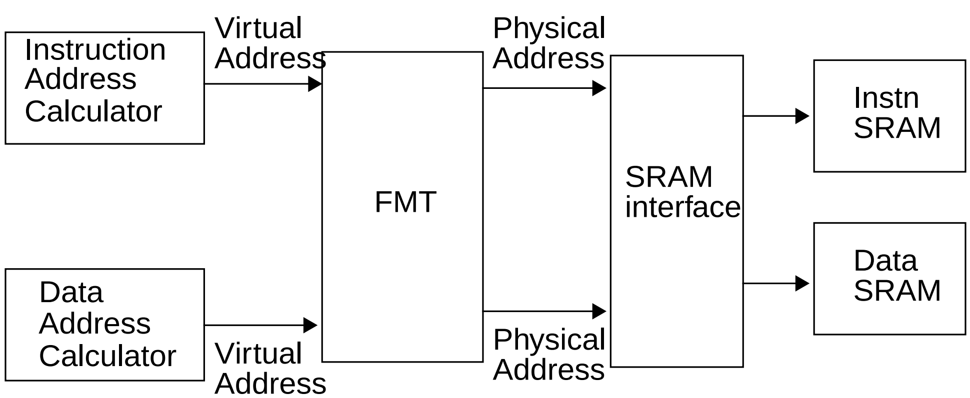


Figure 1-2 Address Translation During a Cache Access

1.2.1.5 Cache Controllers

The data and instruction cache controllers support caches of various sizes, organizations and set associativities. For example, the data cache can be 2 Kbytes in size and 2-way set associative, while the instruction cache can be 8 Kbytes in size and 4-way set associative. There is a separate cache controller for the instruction cache and the data cache.

Each cache controller contains and manages a one-line fill buffer. Besides accumulating data to be written to the cache, the fill buffer is accessed in parallel with the cache and data can be bypassed back to the core.

Refer to Chapter 7, “Caches,” on page 153 for more information on the instruction and data cache controllers.

1.2.1.6 Bus Interface Unit (BIU)

The Bus Interface Unit (BIU) controls the external interface signals. Additionally, it contains the implementation of a 32-byte collapsing write buffer. The purpose of this buffer is to hold and combine write transactions before issuing them to the external interface. Since the data caches for all cores follow a write-through cache policy, the write buffer significantly reduces the number of write transactions on the external interface as well as reducing the amount of stalling in the core due to issuance of multiple writes in a short period of time.

The write buffer is organized as two 16-byte buffers. Each buffer contains data from a single 16-byte aligned block of memory. One buffer contains the data currently being transferred on the external interface, while the other buffer contains accumulating data from the core.

1.2.1.7 Power Management

The core offers a number of power management features, including low-power design, active power management, and power-down modes of operation. The core is a static design that supports a WAIT instruction designed to signal the rest of the device that execution and clocking should be halted, hence reducing system power consumption during idle periods.

The core provides two mechanisms for system-level, low-power support:

- Register-controlled power management
- Instruction-controlled power management

In register-controlled power management mode the core provides three bits in the CP0 Status register for software control of the power management function and allows interrupts to be serviced even when the core is in power-down mode. In instruction-controlled power-down mode execution of the WAIT instruction is used to invoke low-power mode.

Refer to Chapter 8, “Power Management,” on page 163 for more information on power management.

1.2.2 Optional Logic Blocks

The core consists of the following optional logic blocks as shown in the block diagram in Figure 1-1 on page 5.

1.2.2.1 MIPS16e Application Specific Extension

The 4KE core includes optional support for the MIPS16e ASE. This ASE improves code density through the use of 16-bit encodings of MIPS32 instructions plus some MIPS16e-specific instructions. PC relative loads allow quick access to constants. Save/Restore macro instructions provide for single instruction stack frame setup/teardown for efficient subroutine entry/exit. Sign- and zero-extend instructions improve handling of 8bit and 16bit datatypes.

A decompressor converts the MIPS16e 16-bit instructions fetched from the instruction cache or external interface back into 32-bit instructions for execution by the core.

1.2.2.2 Instruction Cache

The instruction cache is an optional on-chip memory array of up to 64 Kbytes. The cache is virtually indexed and physically tagged, allowing the virtual-to-physical address translation to occur in parallel with the cache access rather than having to wait for the physical address translation. The tag holds 22 bits of the physical address, a valid bit, and a lock bit. There is a separate tag array which holds data used in the Least Recently Used (LRU) replacement scheme. The LRU array ranges from 0-6 bits depending on associativity.

All cores support instruction cache locking. Cache locking allows critical code to be locked into the cache on a “per-line” basis, enabling the system designer to maximize the efficiency of the system cache. Cache locking is always available on all instruction cache entries. Entries can be marked as locked or unlocked (by setting or clearing the lock bit) on a per-entry basis using the CACHE instruction.

The LRU array must be bit-writable. The tag and data arrays only need to be word-writable.

1.2.2.3 Data Cache

The data cache is an optional on-chip memory array of up to 64 Kbytes. The cache is virtually indexed and physically tagged, allowing the virtual-to-physical address translation to occur in parallel with the cache access. The tag holds 22 bits of the physical address, a valid bit, and a lock bit. A separate array holds the dirty and LRU bits; this array ranges from 0-10 bits depending on the associativity.

In addition to instruction cache locking, all cores also support a data cache locking mechanism identical to the instruction cache, with critical data segments to be locked into the cache on a “per-line” basis. The locked contents cannot be selected for replacement on a cache miss, but can be updated on a store hit.

Cache locking is always available on all data cache entries. Entries can be marked as locked or unlocked on a per-entry basis using the CACHE instruction.

The physical data cache memory must be byte writable to support sub-word store operations. The LRU/dirty bit array must be bit-writable.

1.2.2.4 EJTAG Controller

All cores provide basic EJTAG support with debug mode, run control, single step and software breakpoint instruction (SDBBP) as part of the core. These features allow for the basic software debug of user and kernel code.

Optional EJTAG features include hardware breakpoints. A 4KE core may have four instruction breakpoints and two data breakpoints, two instruction breakpoints and one data breakpoint, or no breakpoints. The hardware instruction breakpoints can be configured to generate a debug exception when an instruction is executed anywhere in the virtual address space. Bit mask and Address Space Identifier (ASID) values may apply in the address compare. These breakpoints are not limited to code in RAM like the software instruction breakpoint (SDBBP). The data breakpoints can be configured to generate a debug exception on a data transaction. The data transaction may be qualified with both virtual address, data value, size and load/store transaction type. Bit mask and ASID values may apply in the address compare, and byte mask may apply in the value compare.

An optional TAP, enabling communication between an EJTAG probe and the CPU through a dedicated port, may also be applied to the core. This provides the possibility for debugging without debug code in the application, and for download of application code to the system.

Another optional block is EJTAG Trace which enables real-time tracing capability. The trace information can be stored to either an on-chip trace memory, or to an off-chip trace probe. The trace of program flow is highly flexible and can include instruction program counter as well as data addresses and data values. The trace features provides a powerful software debugging mechanism.

Refer to Chapter 9, “EJTAG Debug Support,” on page 165 for more information on the EJTAG features.

1.2.2.5 Coprocessor 2 Interface (CP2)

The optional coprocessor 2 (CP2) interface provides a full-featured interface for a coprocessor. It provides full support for all the MIPS32 COP2 instructions, with the exception of the 64-bit Load/Store instructions (LDC2/SDC2).

The CP2 interface can provide access to a graphics accelerator coprocessor or a simple register file. There is no support for the floating-point coprocessor COP1, which requires 64-bit data transfers.

Refer to Chapter 11, “MIPS32™ 4KE™ Processor Core Instructions,” on page 237 for more information on the Coprocessor 2 supported instructions.

1.2.2.6 CorExtend™ User Defined Instructions (UDI)

This optional module contains (if implemented) support for CorExtend user defined instructions. These instructions must be defined at build-time for the 4KE core. Access to UDI requires a separate license from MIPS, and the core is then referred to as the 4KE Pro™ core. When licensed, 16 instructions in the opcode map are available for UDI, and each instruction can have single or multi-cycle latency. A UDI instruction can operate on any one or two general-purpose registers or immediate data contained within the instruction, and must always write the result of each instruction back to a general purpose register. Implementation details for UDI can be found in other documents available from MIPS.

Refer to Section 11-3, "Special2 Opcode Encoding of Function Field" for a specification of the opcode map available for user defined instructions.

Chapter 2

Pipeline

The MIPS32™ 4KE™ processor core implements a 5-stage pipeline similar to the original R3000 pipeline. The pipeline allows the processor to achieve high frequency while minimizing device complexity, reducing both cost and power consumption. This chapter contains the following sections:

2.1 Pipeline Stages

The pipeline consists of five stages:

- Instruction (I stage)
- Execution (E stage)
- Memory (M stage)
- Align (A stage)
- Writeback (W stage)

A 4KE core implements a “Bypass” mechanism that allows the result of an operation to be sent directly to the instruction that needs it without having to write the result to the register and then read it back.

Figure 2-1 on page 12 shows the operations performed in each pipeline stage of the 4KE processor.

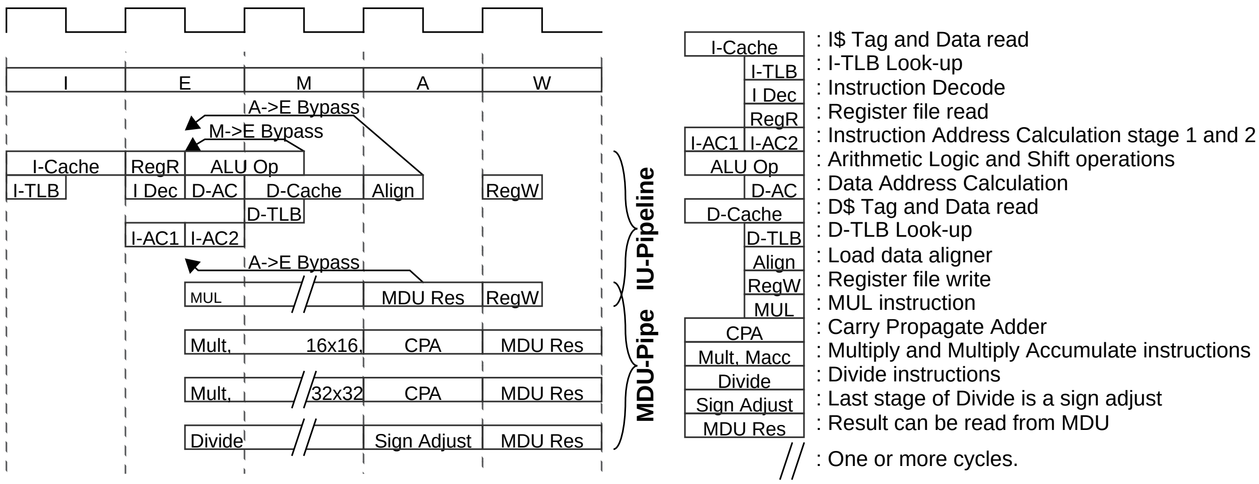


Figure 2-1 4KE Core Pipeline Stages

Figure 2-2 shows the operations performed in each pipeline stage of the 4KEm processor core.

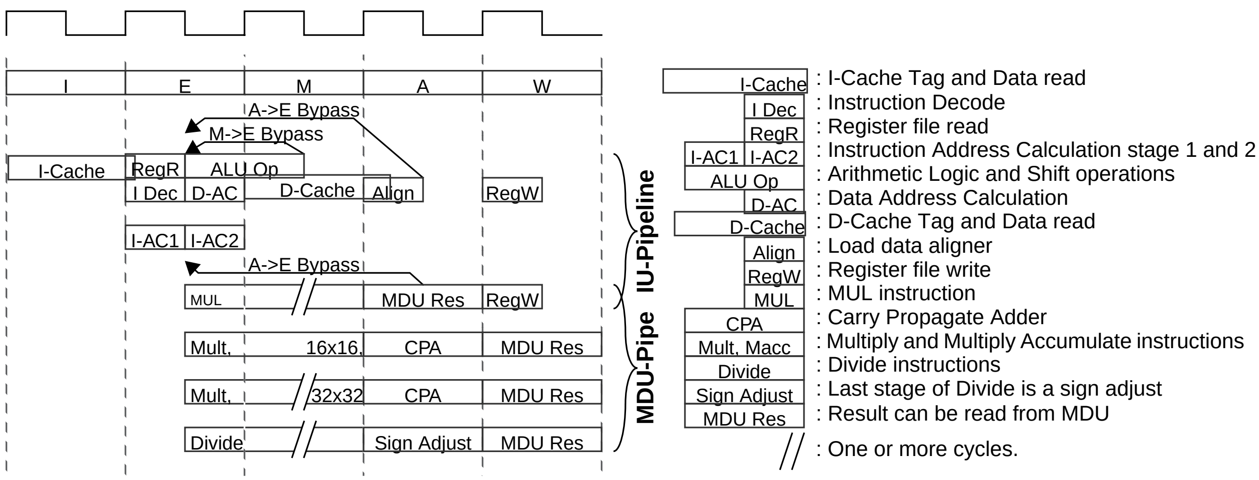


Figure 2-2 4KEm Core Pipeline Stages

Figure 2-3 shows the operations performed in each pipeline stage of the 4KEp processor core.

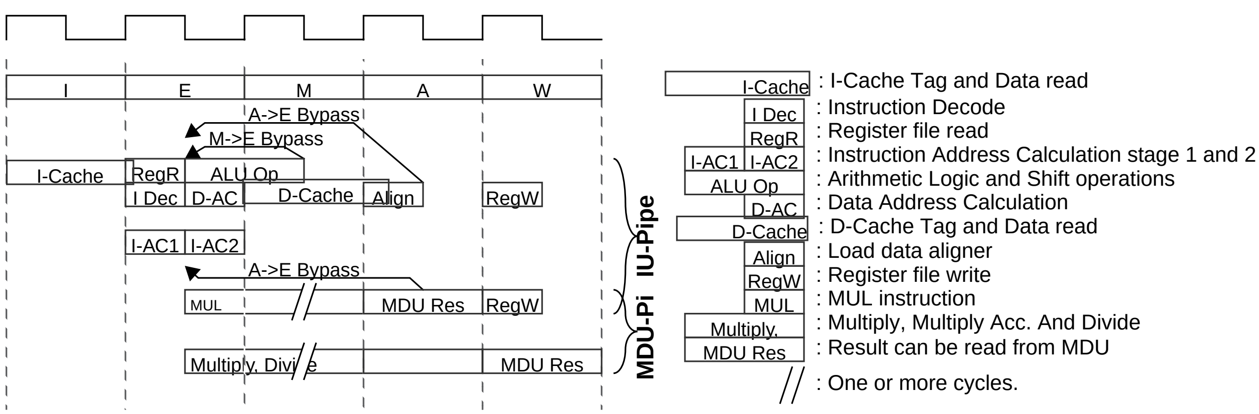


Figure 2-3 4KEp Core Pipeline Stages

2.1.1 I Stage: Instruction Fetch

During the Instruction fetch stage:

- An instruction is fetched from the instruction cache.
- The I-TLB performs a virtual-to-physical address translation (4KEc core only).
- MIPS16e instructions are converted into MIPS32-like instructions.

2.1.2 E Stage: Execution

During the Execution stage:

- Operands are fetched from the register file.
- Operands from the M and A stage are bypassed to this stage.
- The Arithmetic Logic Unit (ALU) begins the arithmetic or logical operation for register-to-register instructions.
- The ALU calculates the data virtual address for load and store instructions.
- The ALU determines whether the branch condition is true and calculates the virtual branch target address for branch instructions.
- Instruction logic selects an instruction address.
- All multiply and divide operations begin in this stage.

2.1.3 M Stage: Memory Fetch

During the Memory Fetch stage:

- The arithmetic or logic ALU operation completes.
- The data cache access and the data virtual-to-physical address translation are performed for load and store instructions.
- Data TLB (4KEc core only) and data cache lookup are performed and a hit/miss determination is made.
- A 16x16 or 32x16 MUL operation completes in the array and stalls for one clock in the M stage to complete the carry-propagate-add in the M stage (4KEc and 4KEm cores).
- A 32x32 MUL operation stalls for two clocks in the M stage to complete the second cycle of the array and the carry-propagate-add in the M stage (4KEc and 4KEm cores).
- A multiply operation stalls the MDU pipeline for 31 cycles in the M stage (4KEp core).
- Multiply and divide calculations proceed in the MDU. If the calculation completes before the IU moves the instruction past the M stage, then the MDU holds the result in a temporary register until the IU moves the instructions to the A stage (and it is consequently known that it won't be killed).

2.1.4 A Stage: Align

During the Align stage:

- A separate aligner aligns loaded data with its word boundary.
- A MUL operation makes the result available for writeback. The actual register writeback is performed in the W stage (all 4KE cores).
- From this stage load data or a result from the MDU are available in the E stage for bypassing.

2.1.5 W Stage: Writeback

During the Writeback stage:

- For register-to-register or load instructions, the result is written back to the register file.

2.2 Instruction Cache Miss

When the instruction cache is indexed, the instruction address is translated to determine if the required instruction resides in the cache. An instruction cache miss occurs when the requested instruction address does not reside in the instruction cache. When a cache miss is detected in the I stage, the core transitions to the E stage. The pipeline stalls in the E stage until the miss is resolved. The bus interface unit must select the address from multiple sources. If the address bus is busy, the request will remain in this arbitration stage (B-ASel in Figure 2-4 on page 14) until the bus is available. The core drives the selected address onto the bus. The number of clocks before data is returned is then determined by the array containing the data.

Once the data is returned to the core, the critical word is written to the instruction register for immediate use. The bypass mechanism allows the core to use the data as soon as it arrives, as opposed to having the entire cache line written to the instruction cache, then reading out the required word.

Figure 2-4 on page 14 shows a timing diagram of an instruction cache miss.

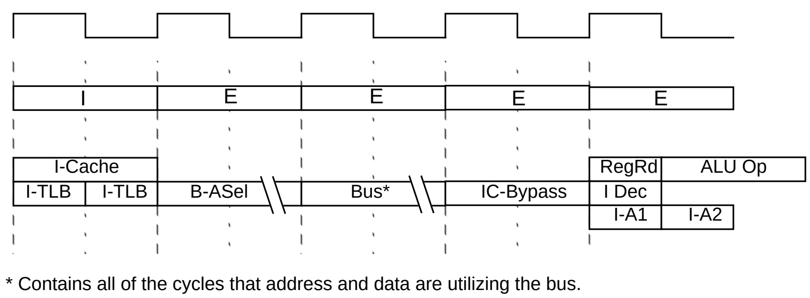


Figure 2-4 Data Cache Miss Timing

2.3 Data Cache Miss

When the data cache is indexed, the data address is translated to determine if the required data resides in the cache. A data cache miss occurs when the requested data address does not reside in the data cache.

When a data cache miss is detected in the M stage (D-TLB), the core transitions to the A stage. The pipeline stalls in the A stage until the miss is resolved (requested data is returned). The bus interface unit arbitrates between multiple requests and selects the correct address to be driven onto the bus (B-ASel in Figure 2-5 on page 15). The core drives the selected address onto the bus. The number of clocks before data is returned is then determined by the array containing the data.

Once the data is returned to the core, the critical word of data passes through the aligner before being forwarded to the execution unit. The bypass mechanism allows the core to use the data as soon as it arrives, as opposed to having the entire cache line written to the data cache, then reading out the required word.

Figure 2-5 on page 15 shows a timing diagram of a data cache miss.

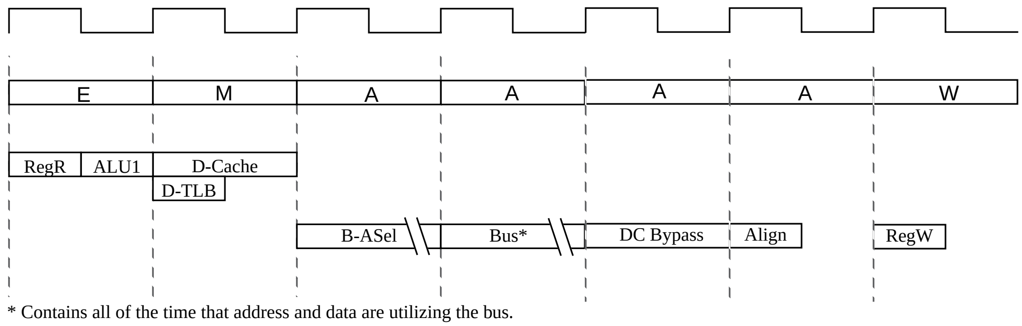


Figure 2-5 Load/Store Cache Miss Timing

2.4 Multiply/Divide Operations

The 4KE core implement the standard MIPS II™ multiply and divide instructions. Additionally, several new instructions were standardized in the MIPS32 architecture for enhanced performance.

The targeted multiply instruction, MUL, specifies that multiply results be placed in the general purpose register file instead of the HI/LO register pair. By avoiding the explicit MFLO instruction, required when using the LO register, and by supporting multiple destination registers, the throughput of multiply-intensive operations is increased.

Four instructions, multiply-add (MADD), multiply-add-unsigned (MADDU) multiply-subtract (MSUB), and multiply-subtract-unsigned (MSUBU), are used to perform the multiply-accumulate and multiply-subtract operations. The MADD/MADDU instruction multiplies two numbers and then adds the product to the current contents of the HI and LO registers. Similarly, the MSUB/MSUBU instruction multiplies two operands and then subtracts the product from the HI and LO registers. The MADD/MADDU and MSUB/MSUBU operations are commonly used in DSP algorithms.

All multiply operations (except the MUL instruction) write to the HI/LO register pair. All integer operations write to the general purpose registers (GPR). Because MDU operations write to different registers than integer operations, following integer instructions can execute before the MDU operation has completed. The MFLO and MFHI instructions are used to move data from the HI/LO register pair to the GPR file. If a MFLO or MFHI instruction is issued before the MDU operation completes, it will stall to wait for the data.

2.5 MDU Pipeline (4KEc and 4KEm Cores)

The 4KEc and 4KEm processor cores contain an autonomous multiply/divide unit (MDU) with a separate pipeline for multiply and divide operations. This pipeline operates in parallel with the integer unit (IU) pipeline and does not stall when the IU pipeline stalls. This allows multi-cycle MDU operations, such as a divide, to be partially masked by system stalls and/or other integer unit instructions.

The MDU consists of a 32x16 booth encoded multiplier array, a carry propagate adder, result/accumulation registers (HI and LO), multiply and divide state machines, and all necessary multiplexers and control logic. The first number shown (‘32’ of 32x16) represents the *rs* operand. The second number (‘16’ of 32x16) represents the *rt* operand. The core only checks the latter *(rt)* operand value to determine how many times the operation must pass through the multiplier array. The 16x16 and 32x16 operations pass through the multiplier array once. A 32x32 operation passes through the multiplier array twice.

The MDU supports execution of a 16x16 or 32x16 multiply operation every clock cycle; 32x32 multiply operations can be issued every other clock cycle. Appropriate interlocks are implemented to stall the issue of back-to-back 32x32

multiply operations. Multiply operand size is automatically determined by logic built into the MDU. Divide operations are implemented with a simple 1 bit per clock iterative algorithm with an early in detection of sign extension on the dividend *(rs)*. Any attempt to issue a subsequent MDU instruction while a divide is still active causes an IU pipeline stall until the divide operation is completed.

Table 2-1 lists the latencies (number of cycles until a result is available) for multiply and divide instructions. The latencies are listed in terms of pipeline clocks. In this table 'latency' refers to the number of cycles necessary for the first instruction to produce the result needed by the second instruction.

Table 2-1 4KEc and 4KEm Core MDU Instruction Latencies

Size of Operand 1st Instruction[1]	Instruction Sequence		Latency Clocks
	1st Instruction	2nd Instruction	
16 bit	MULT/MULTU, MADD/MADDU or MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU or MFHI/MFLO	1
32 bit	MULT/MULTU, MADD/MADDU, or MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU or MFHI/MFLO	2
16 bit	MUL	Integer operation[2]	2[3]
32 bit	MUL	Integer operation[2]	2[3]
8 bit	DIVU	MFHI/MFLO	9
16 bit	DIVU	MFHI/MFLO	17
24 bit	DIVU	MFHI/MFLO	25
32 bit	DIVU	MFHI/MFLO	33
8 bit	DIV	MFHI/MFLO	10[4]
16 bit	DIV	MFHI/MFLO	18[4]
24 bit	DIV	MFHI/MFLO	26[4]
32 bit	DIV	MFHI/MFLO	34[4]
any	MFHI/MFLO	Integer operation[2]	2
any	MTHI/MTLO	MADD/MADDU or MSUB/MSUBU	1

Note: [1] For multiply operations, this is the *rt* operand. For divide operations, this is the *rs* operand.

Note: [2] Integer Operation refers to any integer instruction that uses the result of a previous MDU operation.

Note: [3] This does not include the 1 or 2 IU pipeline stalls (16 bit or 32 bit) that the MUL operation causes irrespective of the following instruction.These stalls do not add to the latency of 2.

Note: [4] If both operands are positive, then the Sign Adjust stage is bypassed. Latency is then the same as for DIVU.

In Table 2-1 a latency of one means that the first and second instructions can be issued back to back in the code without the MDU causing any stalls in the IU pipeline. A latency of two means that if issued back to back, the IU pipeline will be stalled for one cycle. MUL operations are special because it needs to stall the IU pipeline in order to maintain its register file write slot. Consequently the MUL 16x16 or 32x16 operation will always force a one cycle stall of the IU pipeline, and the MUL 32x32 will force a two cycle stall. If the integer instruction immediately following the MUL operation uses its result, an additional stall is forced on the IU pipeline.

Table 2-2 lists the repeat rates (peak issue rate of cycles until the operation can be reissued) for multiply accumulate/subtract instructions. The repeat rates are listed in terms of pipeline clocks. In this table 'repeat rate' refers to the case where the first MDU instruction (in the table below) if back-to-back with the second instruction.

Table 2-2 4KEc Core MDU Instruction Repeat Rates

Operand Size of 1st Instruction	Instruction Sequence		Repeat Rate
	1st Instruction	2nd Instruction	
16 bit	MULT/MULTU, MADD/MADDU, MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU	1
32 bit	MULT/MULTU, MADD/MADDU, MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU	2

Figure 2-6 below shows the pipeline flow for the following sequence:

1. 32x16 multiply ($Mult_1$)
2. Add
3. 32x32 multiply ($Mult_2$)
4. Subtract (Sub)

The 32x16 multiply operation requires one clock of each pipeline stage to complete. The 32x32 multiply operation requires two clocks in the M_{MDU} pipe-stage. The MDU pipeline is shown as the shaded areas of Figure 2-6 and always starts a computation in the final phase of the E stage. As shown in the figure, the M_{MDU} pipe-stage of the MDU pipeline occurs in parallel with the M stage of the IU pipeline, the A_{MDU} stage occurs in parallel with the A stage, and the W_{MDU} stage occurs in parallel with the W stage. In general this need not be the case. Following the 1st cycle of the M stages, the two pipelines need not be synchronized. This does not present a problem because results in the MDU pipeline are written to the HI and LO registers, while the integer pipeline results are written to the register file.

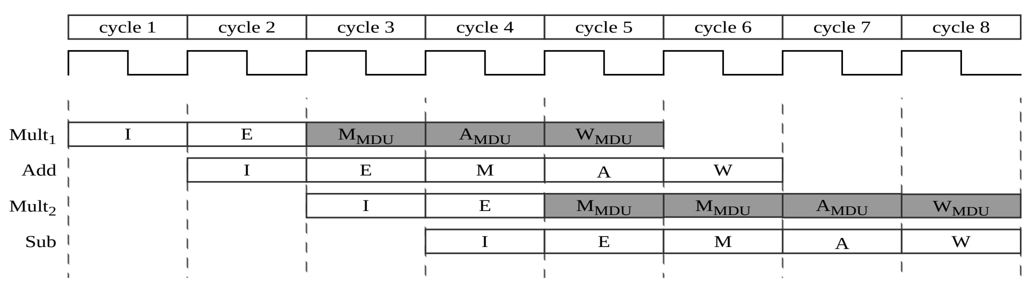


Figure 2-6 MDU Pipeline Behavior During Multiply Operations (4KEc & 4KEm Processors)

The following is a cycle-by-cycle analysis of Figure 2-6.

1. The first 32x16 multiply operation ($Mult_1$) is fetched from the instruction cache and enters the I stage.
2. An Add operation enters the I stage. The $Mult_1$ operation enters the E stage. The integer and MDU pipelines share the I and E pipeline stages. At the end of the E stage in cycle 2, the MDU pipeline starts processing the multiply operation ($Mult_1$).
3. In cycle 3 a 32x32 multiply operation ($Mult_2$) enters the I stage and is fetched from the instruction cache. Since the Add operation has not yet reached the M stage by cycle 3, there is no activity in the M stage of the integer pipeline at this time.

4. In cycle 4 the Subtract instruction enters I stage. The second multiply operation ($Mult_2$) enters the E stage. And the Add operation enters M stage of the integer pipe. Since the $Mult_1$ multiply is a 32x16 operation, only one clock is required for the M_{MDU} stage, hence the $Mult_1$ operation passes to the A_{MDU} stage of the MDU pipeline.
5. In cycle 5 the Subtract instruction enters E stage. The $Mult_2$ multiply enters the M_{MDU} stage. The Add operation enters the A stage of the integer pipeline. The $Mult_1$ operation completes and is written back in to the HI/LO register pair in the W_{MDU} stage.
6. Since a 32x32 multiply requires two passes through the multiplier, with each pass requiring one clock, the 32x32 $Mult_2$ remains in the M_{MDU} stage in cycle 6. The Sub instruction enters M stage in the integer pipeline. The Add operation completes and is written to the register file in the W stage of the integer pipeline.
7. The $Mult_2$ multiply operation progresses to the A_{MDU} stage, and the Sub instruction progress to the A stage.
8. The $Mult_2$ operation completes and is written to the HI/LO registers pair the W_{MDU} stage, while the Sub instruction write to the register file in the W stage.

2.5.1 32x16 Multiply (4KEc & 4KEm Cores)

The 32x16 multiply operation begins in the last phase of the E stage, which is shared between the integer and MDU pipelines. In the latter phase of the E stage, the *rs* and *rt* operands arrive and the booth-recoding function occurs at this time. The multiply calculation requires one clock and occurs in the M_{MDU} stage. In the A_{MDU} stage, the carry-propagate-add (CPA) function occurs and the operation is completed. The result is ready to be read from the HI/LO registers in the W_{MDU} stage.

Figure 2-7 shows a diagram of a 32x16 multiply operation.

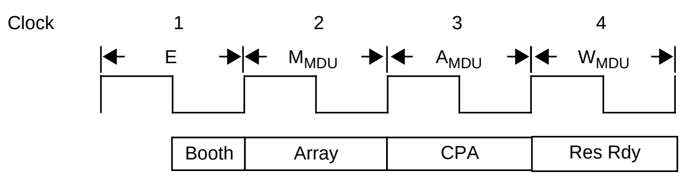


Figure 2-7 MDU Pipeline Flow During a 32x16 Multiply Operation

2.5.2 32x32 Multiply (4KEc & 4KEm Cores)

The 32x32 multiply operation begins in the last phase of the E stage, which is shared between the integer and MDU pipelines. In the latter phase of the E stage, the *rs* and *rt* operands arrive and the booth recoding function occurs at this time. The multiply calculation requires two clocks and occurs in the M_{MDU} stage. In the A_{MDU} stage, the CPA function occurs and the operation is completed.

Figure 2-8 shows a diagram of a 32x32 multiply operation.

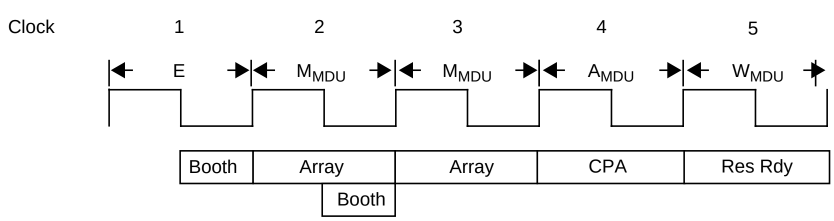


Figure 2-8 MDU Pipeline Flow During a 32x32 Multiply Operation

2.5.3 Divide (4KEc & 4KEm Cores)

Divide operations are implemented using a simple non-restoring division algorithm. This algorithm works only for positive operands, hence the first cycle of the M_{MDU} stage is used to negate the *rs* operand (RS Adjust) if needed. Note that this cycle is spent even if the adjustment is not necessary. During the next maximum 32 cycles (3-34) an iterative add/subtract loop is executed. In cycle 3 an early-in detection is performed in parallel with the add/subtract. The adjusted *rs* operand is detected to be zero extended on the upper most 8, 16 or 24 bits. If this is the case the following 7, 15 or 23 cycles of the add/subtract iterations are skipped.

The remainder adjust (Rem Adjust) cycle is required if the remainder was negative. Note that this cycle is spent even if the remainder was positive. A sign adjust is performed on the quotient and/or remainder if necessary. The sign adjust stage is skipped if both operands are positive. In this case the Rem Adjust is moved to the A_{MDU} stage.

Figure 2-9 on page 19, Figure 2-10 on page 19, Figure 2-11 on page 19 and Figure 2-12 on page 20 show the latency for 8, 16, 24 and 32 bit divide operations, respectively. The repeat rate is either 11, 19, 27 or 35 cycles (one less if the *sign adjust* stage is skipped) as a second divide can be in the *RS Adjust* stage when the first divide is in the *Reg WR* stage.

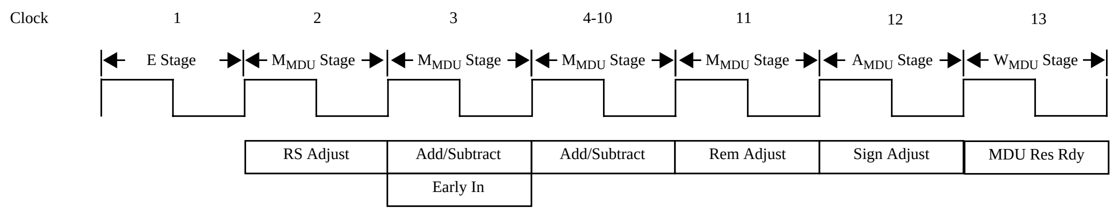


Figure 2-9 MDU Pipeline Flow During a 8-bit Divide (DIV) Operation

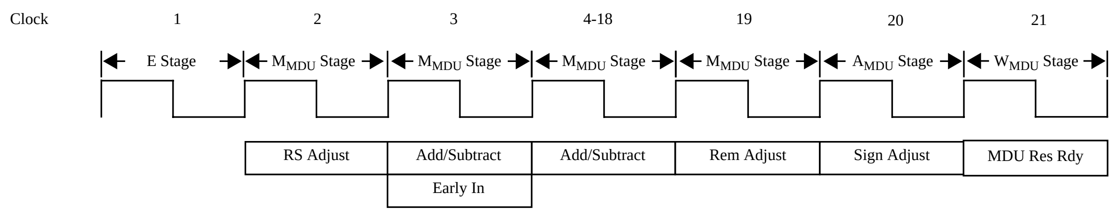


Figure 2-10 MDU Pipeline Flow During a 16-bit Divide (DIV) Operation

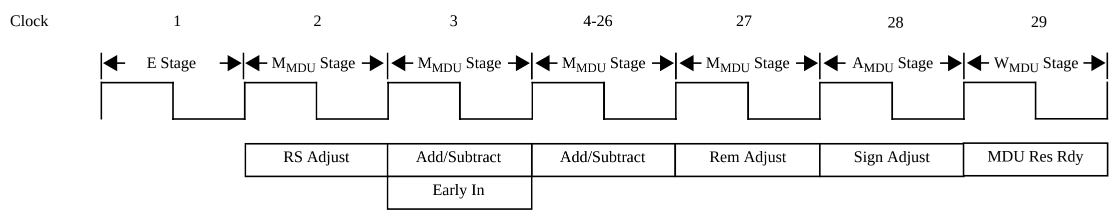


Figure 2-11 MDU Pipeline Flow During a 24-bit Divide (DIV) Operation

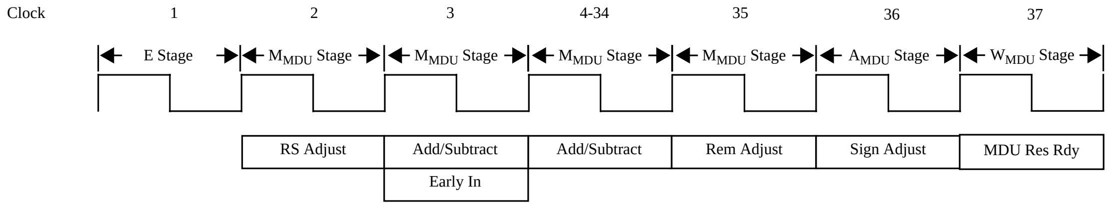


Figure 2-12 MDU Pipeline Flow During a 32-bit Divide (DIV) Operation

2.6 MDU Pipeline (4KEp Core)

The multiply/divide unit (MDU) is a separate autonomous block for multiply and divide operations. The MDU is not pipelined, but rather performs the computations iteratively in parallel with the integer unit (IU) pipeline. It does not stall when the IU pipeline stalls. This allows the long-running MDU operations to be partially masked by system stalls and/or other integer unit instructions.

The MDU consists of one 32-bit adder result-accumulate registers (HI and LO), a combined multiply/divide state machine and all multiplexers and control logic. A simple 1-bit per clock recursive algorithm is used for both multiply and divide operations. Using booth's algorithm all multiply operations complete in 32 clocks. Two extra clocks are needed for multiply-accumulate. The non-restoring algorithm used for divide operations will not work with negative numbers. Adjustment before and after are thus required depending on the sign of the operands. All divide operations complete in 33 to 35 clocks.

Table 2-3 lists the latencies (number of cycles until a result is available) for multiply and divide instructions. The latencies are listed in terms of pipeline clocks. In this table 'latency' refers to the number of cycles necessary for the second instruction to use the results of the first.

Table 2-3 4KEp Core Instruction Latencies

Operand Signs of 1st Instruction (Rs,Rt)	Instruction Sequence		Latency Clocks
	1st Instruction	2nd Instruction	
any, any	MULT/MULTU	MADD/MADDU, MSUB/MSUBU, or MFHI/MFLO	32
any, any	MADD/MADDU, MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU, or MFHI/MFLO	34
any, any	MUL	Integer operation[1]	32
any, any	DIVU	MFHI/MFLO	33
pos, pos	DIV	MFHI/MFLO	33
any, neg	DIV	MFHI/MFLO	34
neg, pos	DIV	MFHI/MFLO	35
any, any	MFHI/MFLO	Integer operation[1]	2
any, any	MTHI/MTLO	MADD/MADDU, MSUB/MSUBU	1

Note: [1] Integer Operation refers to any integer instruction that uses the result of a previous MDU operation.

2.6.1 Multiply (4KEp Core)

Multiply operations are executed using a simple iterative multiply algorithm. Using Booth's approach, this algorithm works for both positive and negative operands. The operation uses 32 cycles in M_{MDU} stage to complete a multiplication. The register writeback to HI and LO are done in the A stage. For MUL operations, the register file writeback is done in the W_{MDU} stage.

Figure 2-13 shows the latency for a multiply operation. The repeat rate is 33 cycles as a second multiply can be in the first M_{MDU} stage when the first multiply is in A_{MDU} stage.

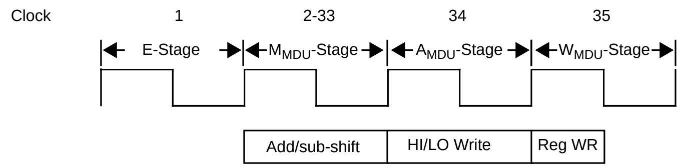


Figure 2-13 4KEp MDU Pipeline Flow During a Multiply Operation

2.6.2 Multiply Accumulate (4KEp Core)

Multiply-accumulate operations use the same multiply machine as used for multiply only. Two extra stages are needed to perform the addition/subtraction. The operations uses 34 cycles in M_{MDU} stage to complete the multiply-accumulate. The register writeback to HI and LO are done in the A stage.

Figure 2-14 shows the latency for a multiply-accumulate operation. The repeat rate is 35 cycles as a second multiply-accumulate can be in the E stage when the first multiply is in the last M_{MDU} stage.

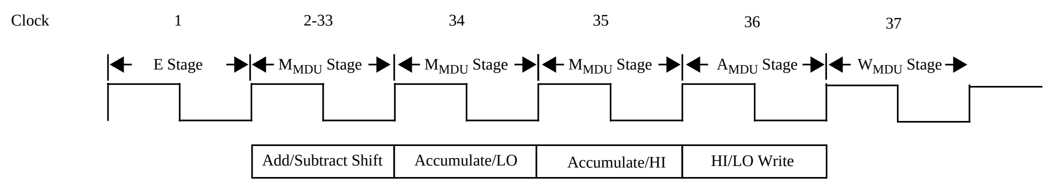


Figure 2-14 4KEp MDU Pipeline Flow During a Multiply Accumulate Operation

2.6.3 Divide (4KEp Core)

Divide operations also implement a simple non-restoring algorithm. This algorithm works only for positive operands, hence the first cycle of the M_{MDU} stage is used to negate the rs operand (RS Adjust) if needed. Note that this cycle is executed even if negation is not needed. The next 32 cycle (3-34) executes an interactive add/subtract-shift function.

Two sign adjust (Sign Adjust 1/2) cycles are used to change the sign of one or both the quotient and the remainder. Note that one or both of these cycles are skipped if they are not needed. The rule is, if both operands were positive or if this is an unsigned division; both of the sign adjust cycles are skipped. If the *rs* operand was negative, one of the sign adjust cycles is skipped. If only the *rs* operand was negative, none of the sign adjust cycles are skipped. Register writeback to HI and LO are done in the A stage.

Figure 2-15 shows the pipeline flow for a divide operation. The repeat rate is either 34, 35 or 36 cycles (depending on how many sign adjust cycles are skipped) as a second divide can be in the E stage when the first divide is in the last M_{MDU} stage.

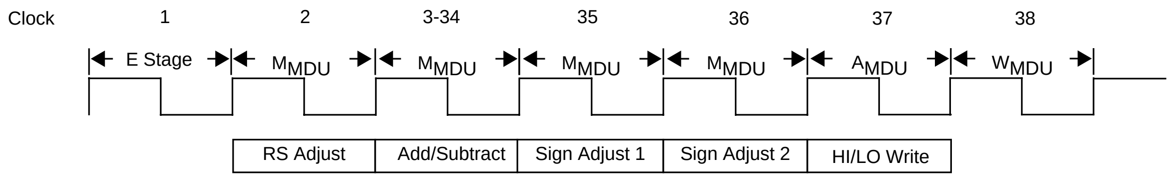


Figure 2-15 4KEp MDU Pipeline Flow During a Divide (DIV) Operation

2.7 Branch Delay

The pipeline has a branch delay of one cycle. The one-cycle branch delay is a result of the branch decision logic operating during the E pipeline stage. This allows the branch target address to be used in the I stage of the instruction following 2 cycles after the branch instruction. By executing the 1st instruction following the branch instruction sequentially before switching to the branch target, the intervening branch delay slot is utilized. This avoids bubbles being injected into the pipeline on branch instructions. Both the address calculation and the branch condition check are performed in the E stage.

The pipeline begins the fetch of either the branch path or the fall-through path in the cycle following the delay slot. After the branch decision is made, the processor continues with the fetch of either the branch path (for a taken branch) or the fall-through path (for the non-taken branch).

The branch delay means that the instruction immediately following a branch is always executed, regardless of the branch direction. If no useful instruction can be placed after the branch, then the compiler or assembler must insert a NOP instruction in the delay slot.

Figure 2-16 illustrates the branch delay.

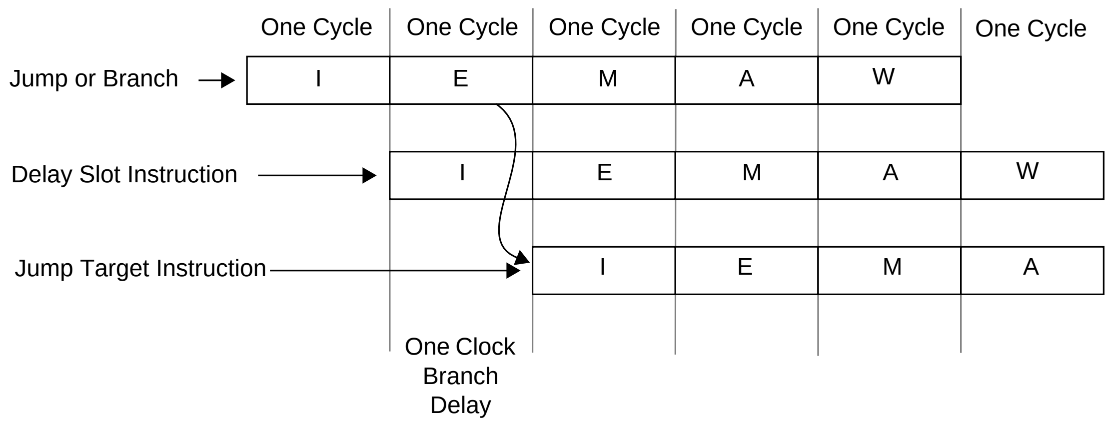


Figure 2-16 IU Pipeline Branch Delay

2.8 Data Bypassing

Most MIPS32 instructions use one or two register values as source operands. These operands are fetched from the register file in the first part of E stage. The ALU straddles the E to M boundary, and can present the result early in M stage. The result is not written to the register file before the W stage however. If no precautions were made, it would take 3 cycles before the result was available for the following instructions. To avoid this, data bypassing is implemented.

Between the register file and the ALU a data bypass multiplexer is placed on both operands (see Figure 2-17 on page 23). This enables the 4KE core to forward data from a preceding instruction whose target is a source register of a following instruction. An M to E bypass and an A to E bypass feed the bypass multiplexers. A W to E bypass is not needed, as the register file is capable of making an internal bypass of Rd write data directly to the Rs and Rt read ports.

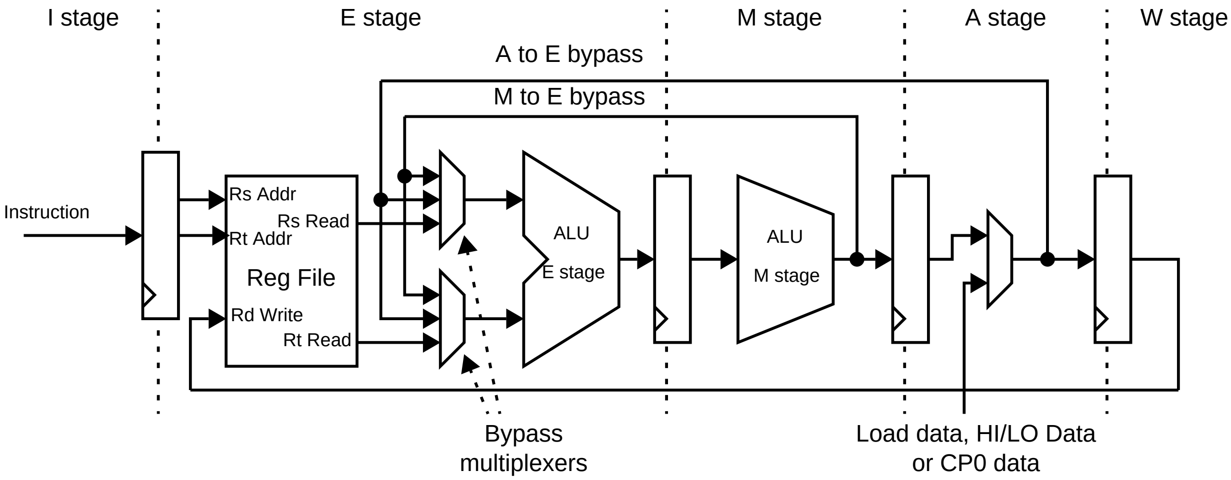


Figure 2-17 IU Pipeline Data bypass

Figure 2-18 on page 23 shows the data bypass for an Add_1 instruction followed by a Sub_2 and another Add_3 instruction. The Sub_2 instruction uses the output from the Add_1 instruction as one of the operands, and thus the M to E bypass is used. The following Add_3 uses the result from both the first Add_1 instruction and the Sub_2 instruction. Since the Add_1 data is now in A stage, the A to E bypass is used, and the M to E bypass is used to bypass the Sub_2 data to the Add_2 instruction.

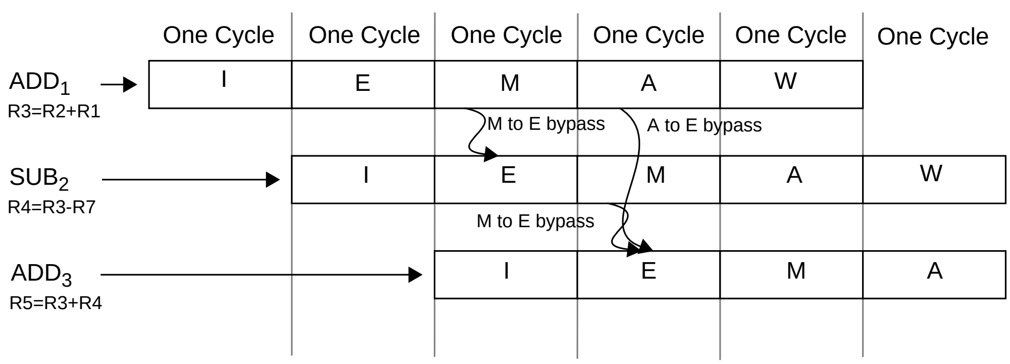


Figure 2-18 IU Pipeline M to E bypass

2.8.1 Load Delay

Load delay refers to the fact, that data fetched by a load instruction is not available in the integer pipeline until after the load aligner in A stage. All instructions need the source operands available in the E stage. An instruction immediately following a load instruction will, if it has the same source register as was the target of the load, cause an instruction interlock pipeline slip in the E stage (see Section 2.12, "Instruction Interlocks" on page 27). If an instruction following the load by 1 or 2 cycles uses the data from the load, the A to E bypass (see Figure 2-17) serves to reduce or avoid stall cycles. An instruction flow of this is shown in Figure 2-19.

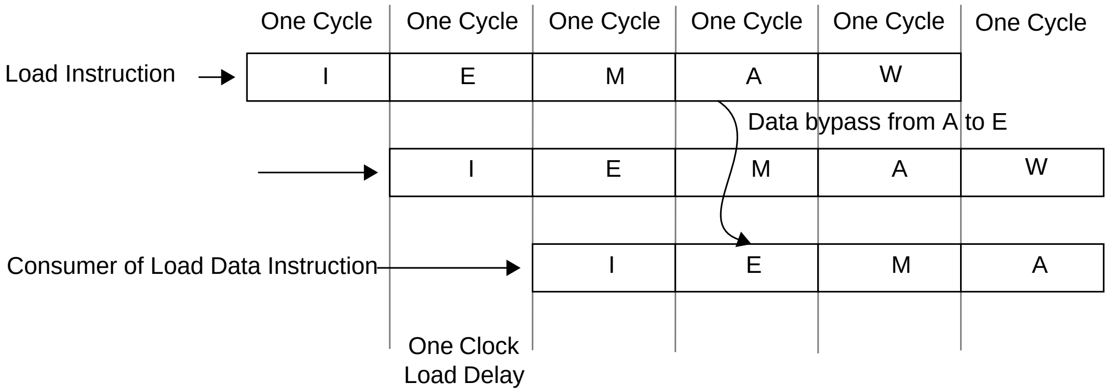


Figure 2-19 IU Pipeline A to E Data bypass

2.8.2 Move from HI/LO and CP0 Delay

As indicated in Figure 2-17, not only load data, but also data moved from the HI or LO registers (MFHI/MFLO) and data moved from CP0 (MFC0) enters the IU-Pipeline in the A stage. That is, data is not available in the integer pipeline until early in the A stage. The A to E bypass is available for this data. But as for Loads, an instruction following immediately after one of these move instructions must be paused for one cycle if the target of the move is among the sources of that following instruction. This then causes an interlock slip in the E stage (see Section 2.12, "Instruction Interlocks" on page 27). An interlock slip after a MFHI is illustrated in Figure 2-20.

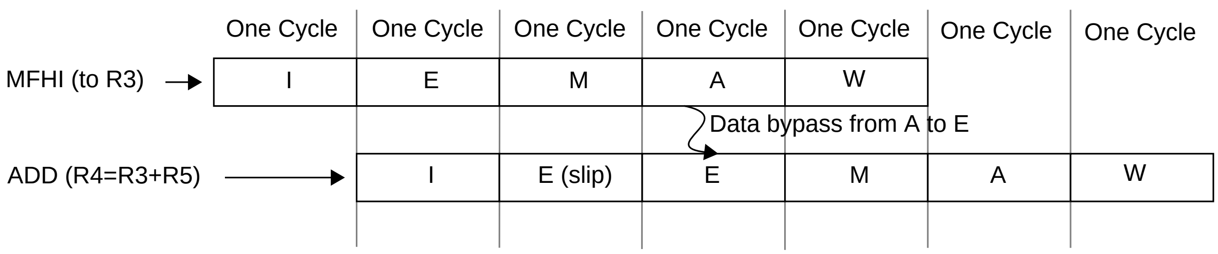


Figure 2-20 IU Pipeline Slip after a MFHI

2.9 Coprocessor 2 instructions

If a coprocessor 2 is attached to the 4KE core, a number of transactions has to take place on the CP2 Interface, for each coprocessor 2 instruction. First of all if the CU[2] bit in the CP0 *Status* register is not set, then no coprocessor 2 related instruction will start a transaction on the CP2 Interface. Rather a Coprocessor Unusable exception will signaled. If the CU[2] bit is set, and a coprocessor 2 instruction is fetched, the following transactions will occur on the CP2 Interface:

1. The Instruction is presented on the instructions bus in E-stage. The coprocessor 2 can do a decode in the same cycle.
2. The Instruction is validated from the core in M-stage. From this point the core will accept control and data signals back from coprocessor 2. All control and data signals from the coprocessor 2 is captured on input latches to the core.

3. If all the expected control and data signals was presented to the core in the previous M-stage, the core will proceed executing the A-stage. If some return information is missing, the A-stage will not advance and cause a slip on all I, E and M-stage, see Section 2.11, "Slip Conditions" on page 26.
   If this instruction involved sending data from the core to the coprocessor 2, then this data is send in A-stage.
4. The instruction completion is signaled to the coprocessor 2 in the W-stage. Potential data from the coprocessor is written in the register file.

Figure 2-21 on page 25 Show the timing relationship between the 4KE core and the coprocessor 2 for all coprocessor 2 instruction.

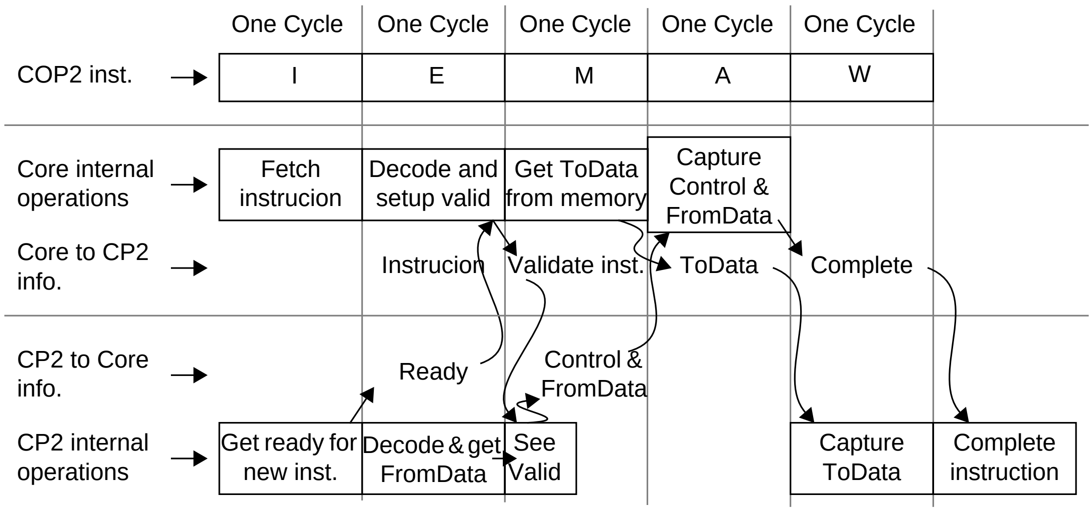


Figure 2-21 Coprocessor 2 Interface Transactions

As can be seen all control and data from the coprocessor must occur in the M-stage. If this is not the case, the A-stage will start slipping in the following cycle, and thus stall the I, E, M and A pipeline stages; but if all expected control and data is available in the M-stage, a Coprocessor 2 instructions can execute with no stalls on the pipeline.

There is only one exception to this, and that is the Branch on Coprocessor conditions (BC2) instruction. All branch instructions, including the regular BEQ, BNE... etc. must be resolved in E-stage. The 4KE core does not have branch prediction logic, and thus the target address must be available before the end of E-stage. The BC2 instruction has to follow the same protocol as all other coprocessor 2 instructions on the CP2 Interface. All core interface operations belonging to the E, M and A stages will have to occur in the E-stage for BC2 instructions. This means that a BC2 instructions always slips for a minimum of 2 cycles in E-stage. Any delay in return of branch information from the Coprocessor 2 will add to the number of slip cycles. All other Coprocessor 2 instructions can operate without slips, provided that all control and data information from the Coprocessor 2 is transferred in the M-stage.

2.10 Interlock Handling

Smooth pipeline flow is interrupted when cache misses occur or when data dependencies are detected. Interruptions handled entirely in hardware, such as cache misses, are referred to as *interlocks*. At each cycle, interlock conditions are checked for all active instructions.

Table 2-4 lists the types of pipeline interlocks for the 4KE processor cores.

Table 2-4 Pipeline Interlocks

Interlock Type	Sources	Slip Stage
ITLB Miss	Instruction TLB	I Stage
ICache Miss	Instruction cache	E Stage
Instruction	Producer-consumer hazards	E/M Stage
	Hardware Dependencies (MDU/TLB)	E Stage
	BC2 waiting for COP2 Condition Check	
DTLB Miss	Data TLB	M Stage
Data Cache Miss	Load that misses in data cache	A Stage
	Multi-cycle cache Op	
	Sync	
	Store when write thru buffer full	
	EJTAG breakpoint on store	
	VA match needing data value comparison	
	Store hitting in fill buffer	
Coprocessor 2 completion slip	Coprocessor 2 control and/or data delay from coprocessor	A Stage

In general, MIPS processors support two types of hardware interlocks:

- Stalls, which are resolved by halting the pipeline
- Slips, which allow one part of the pipeline to advance while another part of the pipeline is held static

In the 4KE processor core, all interlocks are handled as slips.

2.11 Slip Conditions

On every clock internal logic determines whether each pipe stage is allowed to advance. These slip conditions propagate backwards down the pipe. For example, if the M stage does not advance, neither does the E or I stage.

Slipped instructions are retried on subsequent cycles until they issue. The back end of the pipeline advances normally during slips. This resolves the conflict when the slip was caused by a missing result. NOPs are inserted into the bubble in the pipeline. Figure 2-22 on page 27 shows an instruction cache miss.

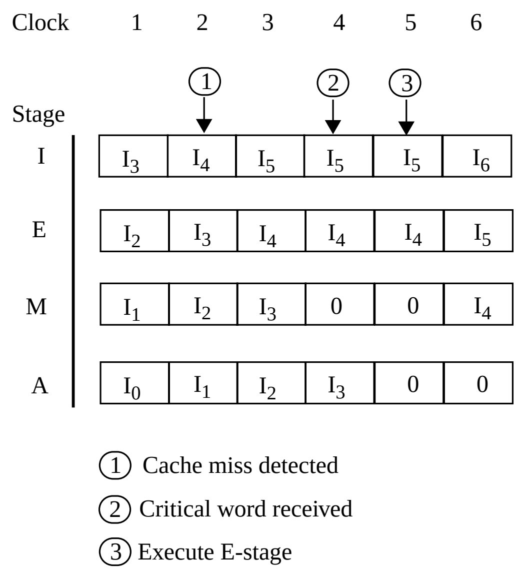


Figure 2-22 Instruction Cache Miss Slip

Figure 2-22 on page 27 shows a diagram of a two-cycle slip. In the first clock cycle, the pipeline is full and the cache miss is detected. Instruction I0 is in the A stage, instruction I1 is in the M stage, instruction I2 is in the E stage, and instruction I3 is in the I stage. The cache miss occurs in clock 2 when the I4 instruction fetch is attempted. I4 advances to the E-stage and waits for the instruction to be fetched from main memory. In this example it takes two clocks (3 and 4) to fetch the I4 instruction from memory. Once the cache miss is resolved in clock 4 and the instruction is bypassed to the E stage, the pipeline is restarted, causing the I4 instruction to finally execute it's E-stage operations.

2.12 Instruction Interlocks

Most instructions can be issued at a rate of one per clock cycle. In order to adhere to the sequential programming model, the issue of an instruction must sometimes be delayed. This to ensure that the result of a prior instruction is available. Table 2-5 details the instruction interactions that prevent an instruction from advancing in the processor pipeline.

Table 2-5 Instruction Interlocks

Instruction Interlocks				
First Instruction		**Second Instruction**	**Issue Delay (in Clock Cycles)**	**Slip Stage**
LB/LBU/LH/LHU/LL/LW/LWL/LWR		Consumer of load data	1	E stage
MFC0		Consumer of destination register	1	E stage
MULTx/MADDx/MSUBx (4KEc and 4KEm cores)	16bx32b	MFLO/MFHI	0	
	32bx32b		1	M stage
MUL (4KEc and 4KEm cores)	16bx32b	Consumer of target data	2	E stage
	32bx32b		3	E stage

Table 2-5 Instruction Interlocks

Instruction Interlocks				
First Instruction		**Second Instruction**	**Issue Delay (in Clock Cycles)**	**Slip Stage**
MUL (4KEc and 4KEm cores)	16bx32b	Non-Consumer of target data	1	E stage
	32bx32b		2	E stage
MFHI/MFLO		Consumer of target data	1	E stage
MULTx/MADDx/MSUBx (4KEc and 4KEm cores)	16bx32b	MULT/MUL/MADD/MSUB MTHI/MTLO/DIV	0[1]	E stage
	32bx32b		1[1]	E stage
DIV		MUL/MULTx/MADDx/ MSUBx/MTHI/MTLO/ MFHI/MFLO/DIV	Until DIV completes	E stage
MULT/MUL/MADD/MSUB/MTHI/MTLO/ MFHI/MFLO/DIV (4KEp core)		MULT/MUL/MADD/MSUB /MTHI/MTLO/MFHI/MFL O/DIV	Until 1st MDU op completes	E stage
MUL (4KEp core)		Any Instruction	Until MUL completes	E stage
MFC0/MFC2/CFC2		Consumer of target data	1	E stage
TLBWR/TLBWI		Load/Store/PREF/CACHE/ COP0 op	2	E stage
TLBR			1	E stage

2.13 Hazards

In general, the 4KE core ensures that instructions are executed following a fully sequential program model. Each instruction in the program sees the results of the previous instruction. There are some deviations to this model. These deviations are referred to as *hazards*.

Prior to Release 2 of the MIPS32™ Architecture, hazards (primarily CP0 hazards) were relegated to implementation-dependent cycle-based solutions, primarily based on the SSNOP instruction. This has been an insufficient and error-prone practice that must be addressed with a firm compact between hardware and software. As such, new instructions have been added to Release 2 of the architecture which act as explicit barriers that eliminate hazards. To the extent that it was possible to do so, the new instructions have been added in such a way that they are backward-compatible with existing MIPS processors.

2.13.1 Types of Hazards

With one exception, all hazards were eliminated in Release 1 of the Architecture for unprivileged software. The exception occurs when unprivileged software writes a new instruction sequence and then wishes to jump to it. Such an operation remained a hazard, and is addressed by the capabilities of Release 2.

In privileged software, there are two different types of hazards: *execution hazards* and *instruction hazards*. Both are defined below.

2.13.1.1 Execution Hazards

Execution hazards are those created by the execution of one instruction, and seen by the execution of another instruction. Table 2-6 lists execution hazards.

Table 2-6 Execution Hazards

Producer	→	Consumer	Hazard On	Spacing (Instructions)
TLBWR, TLBWI	→	TLBP, TLBR	TLB entry	0
		Load/store using new TLB entry	TLB entry	0
MTC0	→	Load/store affected by new state	WatchHi WatchLo	0
LL	→	MFC0	LLAddr	1
MTC0	→	Coprocessor instruction execution depends on the new value of $Status_{CU}$	$Status_{CU}$	1
MTC0	→	ERET	EPC DEPC ErrorEPC	1
MTC0	→	ERET	Status	0
MTC0, EI, DI	→	Interrupted Instruction	$Status_{IE}$	1
MTC0	→	Interrupted Instruction	$Cause_{IP}$	3
TLBR	→	MFC0	EntryHi, EntryLo0, EntryLo1, PageMask	0
TLBP	→	MFC0	Index	0
MTC0	→	TLBR TLBWI TLBWR	EntryHi	1
MTC0	→	TLBP Load/store affected by new state	$EntryHi_{ASID}$	1
MTC0	→	TLBWI TLBWR	EntryLo0 EntryLo1	0
MTC0	→	TLBWI TLBWR	Index	1
MTC0	→	RDPGPR WRPGPR	$SRSCtl_{PSS}$	1
MTC0	→	Instruction not seeing a Timer Interrupt	Compare update that clears Timer Interrupt	4[1]
MTC0	→	Instruction affected by change	Any other CP0 register	2

1. This is the minimum value. Actual value is system-dependent since it is a function of the sequential logic between the *SI_TimerInt* output and the external logic which feeds *SI_TimerInt* back into one of the *SI_Int* inputs, or a function of the method for handling *SI_TimerInt* in an external interrupt controller.

2.13.1.2 Instruction Hazards

Instruction hazards are those created by the execution of one instruction, and seen by the instruction fetch of another instruction. Table 2-7 lists instruction hazards.

Table 2-7 Instruction Hazards

Producer	→	Consumer	Hazard On	Spacing (Instructions)
TLBWR, TLBWI	→	Instruction fetch using new TLB entry	TLB entry	3
MTC0	→	Instruction fetch seeing the new value (including a change to ERL followed by an instruction fetch from the useg segment)	Status	
MTC0	→	Instruction fetch seeing the new value	$\text{EntryHi}_{\text{ASID}}$	3
MTC0	→	Instruction fetch seeing the new value	WatchHi WatchLo	2
Instruction stream write via CACHE	→	Instruction fetch seeing the new instruction stream	Cache entries	3
Instruction stream write via store	→	Instruction fetch seeing the new instruction stream	Cache entries	System-dependent[1]

1. This value depends on how long it takes for the store value to propagate through the system.

2.13.2 Instruction Listing

Table 2-8 lists the instructions designed to eliminate hazards. See the document titled *MIPS32™ Architecture for Programmers: Release 2 Architecture Changes* (MD00232) for a more detailed description of these instructions.

Table 2-8 Hazard Instruction Listing

Mnemonic	Function
EHB	Clear execution hazard
JALR.HB	Clear both execution and instruction hazards
JR.HB	Clear both execution and instruction hazards
SYNCI	Synchronize caches after instruction stream write

2.13.2.1 Instruction Encoding

The EHB instruction is encoded using a variant of the NOP/SSNOP encoding. This encoding was chosen for compatibility with the Release 1 SSNOP instruction, such that existing software may be modified to be compatible with both Release 1 and Release 2 implementations. See the EHB instruction description for additional information.

The JALR.HB and JR.HB instructions are encoding using bit 10 of the *hint* field of the JALR and JR instructions. These encodings were chosen for compatibility with existing MIPS implementations, including many which pre-date the MIPS32 architecture. Because a pipeline flush clears hazards on most early implementations, the JALR.HB or JR.HB instructions can be included in existing software for backward and forward compatibility. See the JALR.HB and JR.HB instructions for additional information.

The SYNCI instruction is encoded using a new encoding of the REGIMM opcode. This encoding was chosen because it causes a Reserved Instruction exception on all Release 1 implementations. As such, kernel software running on processors that don't implement Release 2 can emulate the function using the CACHE instruction.

2.13.3 Eliminating Hazards

The Spacing column shown in Table 2-6 and Table 2-7 indicates the number of unrelated instructions (such as NOPs or SSNOPs) that, prior to the capabilities of Release 2, would need to be placed between the producer and consumer of the hazard in order to ensure that the effects of the first instruction are seen by the second instruction. Entries in the table that are listed as 0 are traditional MIPS hazards which are not hazards on the 4KE core.

With the hazard elimination instructions available in Release 2, the preferred method to eliminate hazards is to place one of the instructions listed in Table 2-8 between the producer and consumer of the hazard. Execution hazards can be removed by using the EHB, JALR.HB, or JR.HB instructions. Instruction hazards can be removed by using the JALR.HB or JR.HB instructions, in conjunction with the SYNCI instruction. Since the 4KE core does not contain caches, the SYNCI instruction is not strictly necessary, but is still recommended to create portable code that can be run on other MIPS processors that may contain caches.

Memory Management

The MIPS32™ 4KE™ processor core includes a Memory Management Unit (MMU) that interfaces between the execution unit and the cache controller. The MIPS32 4KEc core contains a Translation Lookaside Buffer (TLB), while the MIPS32 4KEm and MIPS32 4KEp cores implement a simpler Fixed Mapping (FM) style MMU.

This chapter contains the following sections:

3.1 Introduction

The MMU in a 4KE processor core will translate any virtual address to a physical address before a request is sent to the cache controllers for tag comparison or to the bus interface unit for an external memory reference. This translation is a very useful feature for operating systems when trying to manage physical memory to accommodate multiple tasks active in the same memory, possibly on the same virtual address but of course in different locations in physical memory (4KEc core only). Other features handled by the MMU are protection of memory areas and defining the cache protocol.

In the 4KEc processor core, the MMU is TLB based. The TLB consists of three address translation buffers: a 16 dual-entry fully associative Joint TLB (JTLB), a 4-entry instruction micro TLB (ITLB), and a 4-entry data micro TLB (DTLB). When an address is translated, the appropriate micro TLB (ITLB or DTLB) is accessed first. If the translation is not found in the micro TLB, the JTLB is accessed. If there is a miss in the JTLB, an exception is taken.

In the 4KEm and 4KEp processor cores, the MMU is based on a simple algorithm to translate virtual addresses into physical addresses via a Fixed Mapping (FM) mechanism. These translations are different for various regions of the virtual address space (useg/kuseg, kseg0, kseg1, kseg2/3).

Figure 3-1 shows how the memory management unit interacts with cache accesses in the 4KEc core, while Figure 3-2 shows the equivalent for the 4KEm and 4KEp cores. In the 4KEm and 4KEp cores, note that the FM MMU replaces the ITLB, DTLB and JTLB found in the 4KEc core.

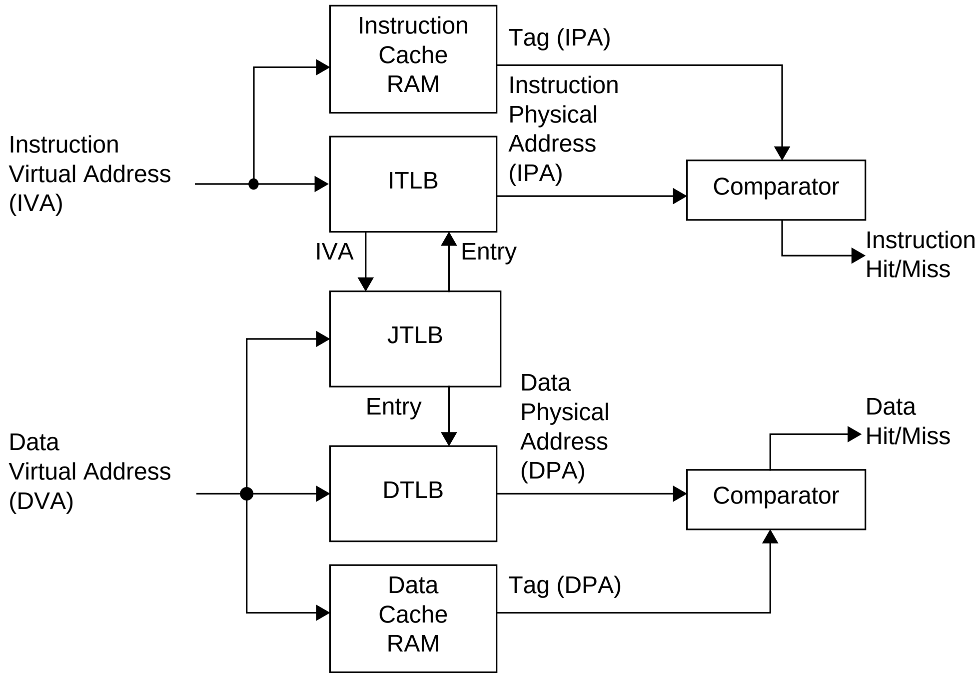


Figure 3-1 Address Translation During a Cache Access in the 4KEc core

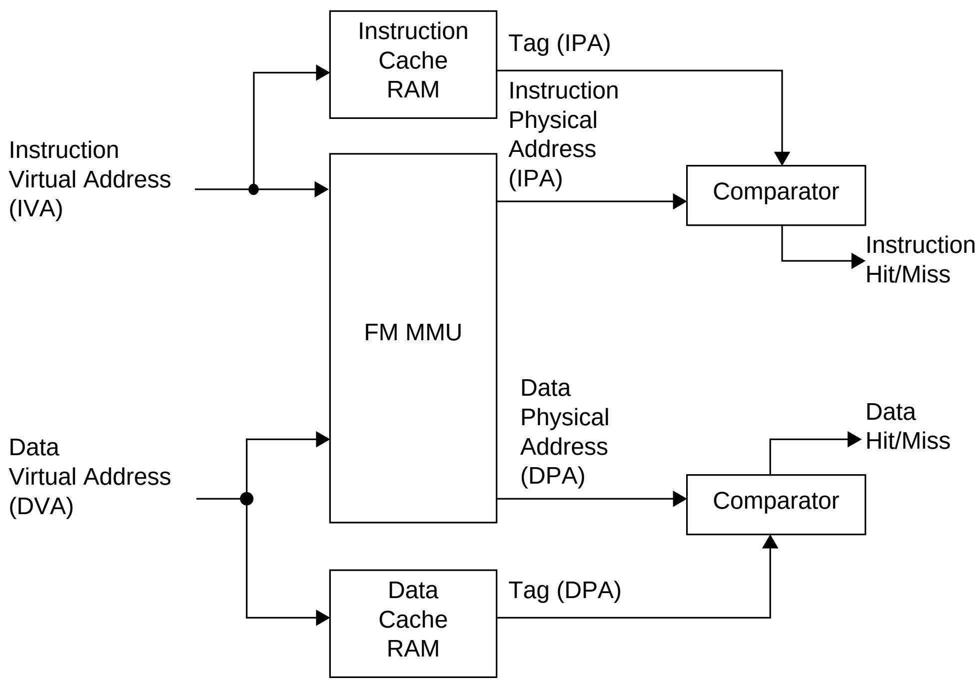


Figure 3-2 Address Translation During a Cache Access in the 4KEm and 4KEp Cores

3.2 Modes of Operation

A 4KE processor core supports three modes of operation:

- User mode
- Kernel mode
- Debug mode

User mode is most often used for application programs. Kernel mode is typically used for handling exceptions and privileged operating system functions, including CP0 management and I/O device accesses. Debug mode is used for software debugging and most likely occurs within a software development tool.

The address translation performed by the MMU depends on the mode in which the processor is operating.

3.2.1 Virtual Memory Segments

The Virtual memory segments are different depending on the mode of operation. Figure 3-3 on page 36 shows the segmentation for the 4 GByte (2^{32} bytes) virtual memory space addressed by a 32-bit virtual address, for the three modes of operation.

The core enters Kernel mode both at reset and when an exception is recognized. While in Kernel mode, software has access to the entire address space, as well as all CP0 registers. User mode accesses are limited to a subset of the virtual address space (0x0000_0000 to 0x7FFF_FFFF) and can be inhibited from accessing CP0 functions. In User mode, virtual addresses 0x8000_0000 to 0xFFFF_FFFF are invalid and cause an exception if accessed.

Debug mode is entered on a debug exception. While in Debug mode, the debug software has access to the same address space and CP0 registers as for Kernel mode. In addition, while in Debug mode the core has access to the debug segment dseg. This area overlays part of the kernel segment kseg3. dseg access in Debug mode can be turned on or off, allowing full access to the entire kseg3 in Debug mode, if so desired.

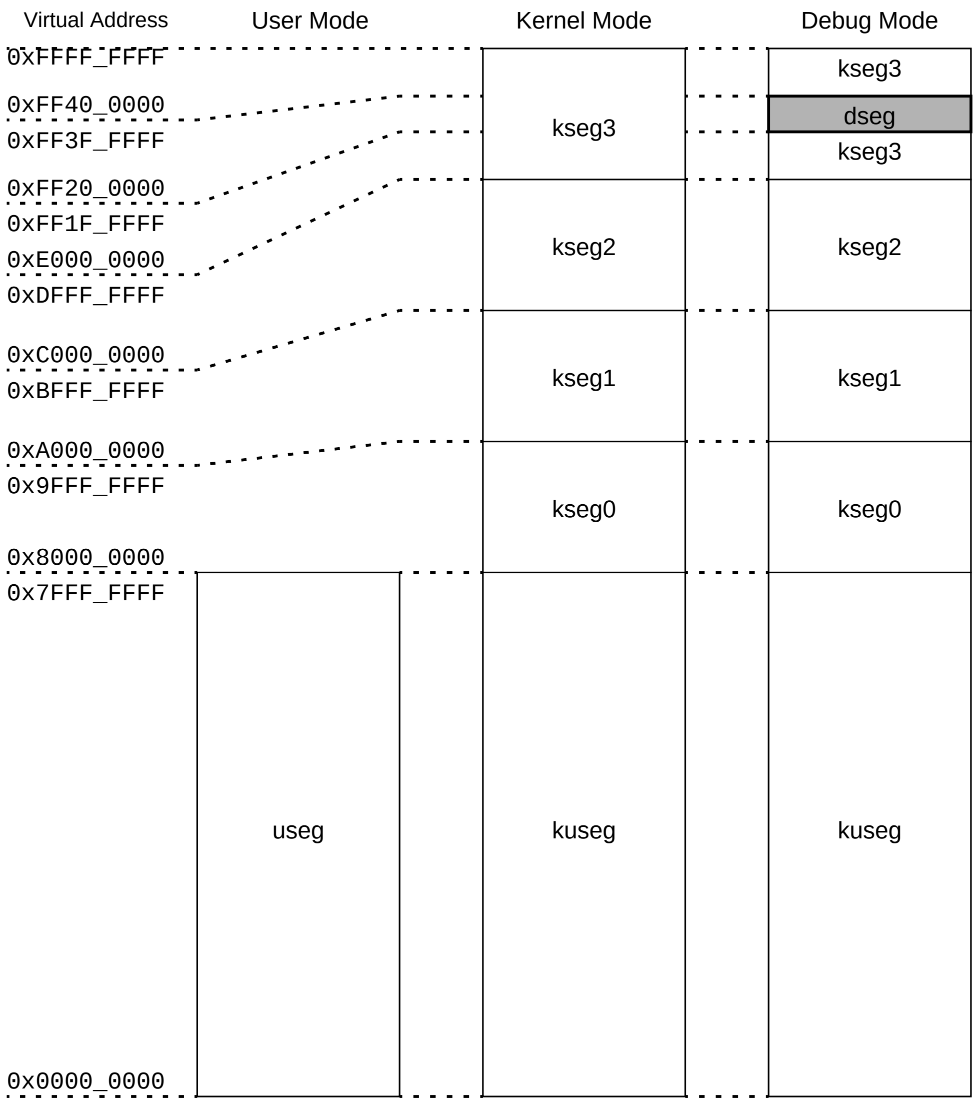


Figure 3-3 4KE processor core Virtual Memory Map.

Each of the segments shown in Figure 3-3 on page 36 are either mapped or unmapped. The following two sub-sections explain the distinction. Then sections Section 3.2.2, "User Mode", Section 3.2.3, "Kernel Mode" and Section 3.2.4, "Debug Mode" specify which segments are actually mapped and unmapped.

3.2.1.1 Unmapped Segments

An unmapped segment does not use the TLB (4KEc core) or the FM (4KEm and 4KEp cores) to translate from virtual-to-physical addresses. Especially after reset, it is important to have unmapped memory segments, because the TLB is not yet programmed to perform the translation.

Unmapped segments have a fixed simple translation from virtual to physical address. This is much like the translations the FM provides for the 4KEm and 4KEp cores, but we will still make the distinction.

Except for kseg0, unmapped segments are always uncached. The cacheability of kseg0 is set in the K0 field of the CP0 register Config (see Section 5.2.21, "Config Register (CP0 Register 16, Select 0)").

3.2.1.2 Mapped Segments

A mapped segment does use the TLB (4KEc core) or the FM (4KEm and 4KEp cores) to translate from virtual-to-physical addresses.

For the 4KEc core, the translation of mapped segments is handled on a per-page basis. Included in this translation is information defining whether the page is cacheable or not, and the protection attributes that apply to the page.

For the 4KEm and 4KEp cores, the mapped segments have a fixed translation from virtual to physical address. The cacheability of the segment is defined in the CP0 register Config, fields K23 and KU (see Section 5.2.21, "Config Register (CP0 Register 16, Select 0)"). Write protection of segments is not possible during FM translation.

3.2.2 User Mode

In user mode, a single 2 GByte (2^{31} bytes) uniform virtual address space called the user segment (useg) is available. Figure 3-4 on page 37 shows the location of user mode virtual address space.

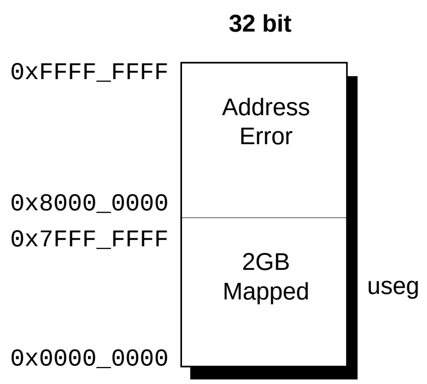


Figure 3-4 User Mode Virtual Address Space

The user segment starts at address 0x0000_0000 and ends at address 0x7FFF_FFFF. Accesses to all other addresses cause an address error exception.

The processor operates in User mode when the *Status* register contains the following bit values:

- UM = 1
- EXL = 0
- ERL = 0

In addition to the above values, the DM bit in the *Debug* register must be 0.

Table 3-1 lists the characteristics of the useg User mode segments.

Table 3-1 User Mode Segments

Address Bit Value	Status Register Bit Value EXL	ERL	UM	Segment Name	Address Range	Segment Size
32-bit A(31) = 0	0	0	1	useg	0x0000_0000 --> 0x7FFF_FFFF	2 GByte (2^{31} bytes)

All valid user mode virtual addresses have their most significant bit cleared to 0, indicating that user mode can only access the lower half of the virtual memory map. Any attempt to reference an address with the most significant bit set while in user mode causes an address error exception.

The system maps all references to *useg* through the TLB (4KEc core) or FM (4KEm and 4KEp cores). For the 4KEc core, the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address before translation. Also for the 4KEc core, bit settings within the TLB entry for the page determine the cacheability of a reference. For the 4KEm and 4KEp cores, the cacheability is set via the KU field of the CP0 Config register.

3.2.3 Kernel Mode

The processor operates in Kernel mode when the DM bit in the *Debug* register is 0 and the *Status* register contains one or more of the following values:

- UM = 0
- ERL = 1
- EXL = 1

When a non-debug exception is detected, EXL or ERL will be set and the processor will enter Kernel mode. At the end of the exception handler routine, an Exception Return (ERET) instruction is generally executed. The ERET instruction jumps to the Exception PC, clears ERL, and clears EXL if ERL=0. This may return the processor to User mode.

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 3-5 on page 39. Also, Table 3-2 lists the characteristics of the Kernel mode segments.

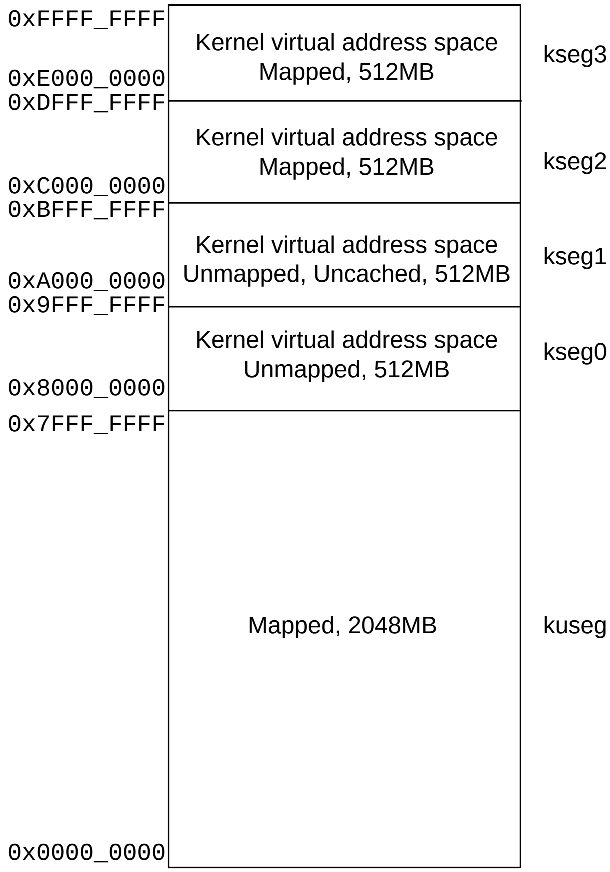


Figure 3-5 Kernel Mode Virtual Address Space

Table 3-2 Kernel Mode Segments

<table>
<tr><th rowspan="2">Address Bit Values</th><th colspan="3">Status Register Is One of These Values</th><th rowspan="2">Segment Name</th><th rowspan="2">Address Range</th><th rowspan="2">Segment Size</th></tr>
<tr><th>UM</th><th>EXL</th><th>ERL</th></tr>
<tr><td>A(31) = 0</td><td colspan="3" rowspan="5">(UM = 0
or
EXL = 1
or
ERL = 1)
and
DM = 0</td><td>kuseg</td><td>0x0000_0000
through
0x7FFF_FFFF</td><td>2 GBytes
(2^{31} bytes)</td></tr>
<tr><td>A(31:29) = 100_2</td><td>kseg0</td><td>0x8000_0000
through
0x9FFF_FFFF</td><td>512 MBytes
(2^{29} bytes)</td></tr>
<tr><td>A(31:29) = 101_2</td><td>kseg1</td><td>0xA000_0000
through
0xBFFF_FFFF</td><td>512 MBytes
(2^{29} bytes)</td></tr>
<tr><td>A(31:29) = 110_2</td><td>kseg2</td><td>0xC000_0000
through
0xDFFF_FFFF</td><td>512 MBytes
(2^{29} bytes)</td></tr>
<tr><td>A(31:29) = 111_2</td><td>kseg3</td><td>0xE000_0000
through
0xFFFF_FFFF</td><td>512 MBytes
(2^{29} bytes)</td></tr>
</table>

3.2.3.1 Kernel Mode, User Space (kuseg)

In Kernel mode, when the most-significant bit of the virtual address (A31) is cleared, the 32-bit kuseg virtual address space is selected and covers the full 2^{31} bytes (2 GBytes) of the current user address space mapped to addresses 0x0000_0000 - 0x7FFF_FFFF. For the 4KEc core, the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

When ERL = 1 in the *Status* register, the user address region becomes a 2^{31}-byte unmapped and uncached address space. While in this setting, the kuseg virtual address maps directly to the same physical address, and does not include the ASID field.

3.2.3.2 Kernel Mode, Kernel Space 0 (kseg0)

In Kernel mode, when the most-significant three bits of the virtual address are 100_2, 32-bit kseg0 virtual address space is selected; it is the 2^{29}-byte (512-MByte) kernel virtual space located at addresses 0x8000_0000 - 0x9FFF_FFFF. References to kseg0 are unmapped; the physical address selected is defined by subtracting 0x8000_0000 from the virtual address. The K0 field of the *Config* register controls cacheability.

3.2.3.3 Kernel Mode, Kernel Space 1 (kseg1)

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are 101_2, 32-bit kseg1 virtual address space is selected. kseg1 is the 2^{29}-byte (512-MByte) kernel virtual space located at addresses 0xA000_0000 - 0xBFFF_FFFF. References to kseg1 are unmapped; the physical address selected is defined by subtracting 0xA000_0000 from the virtual address. Caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) are accessed directly.

3.2.3.4 Kernel Mode, Kernel Space 2 (kseg2)

In Kernel mode, when UM = 0, ERL = 1, or EXL = 1 in the *Status* register, and DM = 0 in the *Debug* register, and the most-significant three bits of the 32-bit virtual address are 110_2, 32-bit kseg2 virtual address space is selected. In the 4KEm and 4KEp processor cores, this 2^{29}-byte (512-MByte) kernel virtual space is located at physical addresses 0xC000_0000 - 0xDFFF_FFFF. In the 4KEc processor core, this space is mapped through the TLB.

3.2.3.5 Kernel Mode, Kernel Space 3 (kseg3)

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are $111_{2\,,}$ the kseg3 virtual address space is selected. In the 4KEm and 4KEp processor cores, this 2^{29}-byte (512-MByte) kernel virtual space is located at physical addresses 0xE000_0000 - 0xFFFF_FFFF. In the 4KEc processor core, this space is mapped through the TLB.

3.2.4 Debug Mode

Debug mode address space is identical to Kernel mode address space with respect to mapped and unmapped areas, except for kseg3. In kseg3, a debug segment dseg co-exists in the virtual address range 0xFF20_0000 to 0xFF3F_FFFF. The layout is shown in Figure 3-6 on page 41.

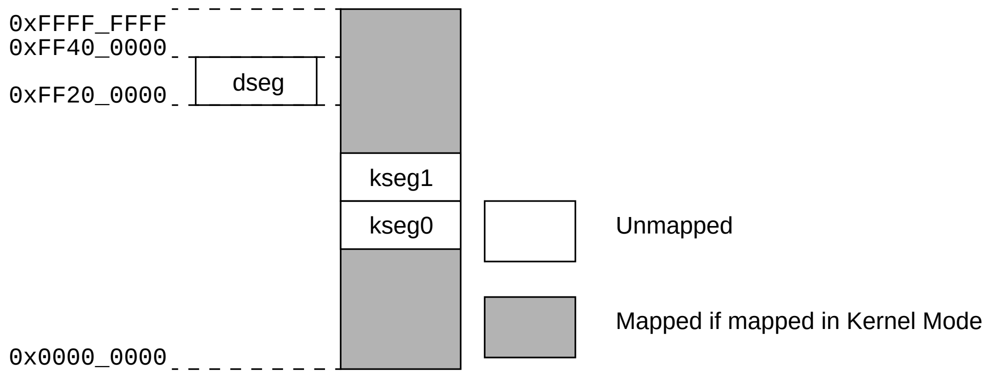


Figure 3-6 Debug Mode Virtual Address Space

The dseg is sub-divided into the dmseg segment at 0xFF20_0000 to 0xFF2F_FFFF which is used when the probe services the memory segment, and the drseg segment at 0xFF30_0000 to 0xFF3F_FFFF which is used when memory-mapped debug registers are accessed. The subdivision and attributes for the segments are shown in Table 3-3.

Accesses to memory that would normally cause an exception if tried from kernel mode cause the core to re-enter debug mode via a debug mode exception. This includes accesses usually causing a TLB exception (4KEc core only), with the result that such accesses are not handled by the usual memory management routines.

The unmapped kseg0 and kseg1 segments from kernel mode address space are available from debug mode, which allows the debug handler to be executed from uncached and unmapped memory.

Table 3-3 Physical Address and Cache Attributes for dseg, dmseg, and drseg Address Spaces

Segment Name	Sub-Segment Name	Virtual Address	Generates Physical Address	Cache Attribute
dseg	dmseg	0xFF20_0000 through 0xFF2F_FFFF	dmseg maps to addresses 0x0_0000 - 0xF_FFFF in EJTAG probe memory space.	Uncached
	drseg	0xFF30_0000 through 0xFF3F_FFFF	drseg maps to the breakpoint registers 0x0_0000 - 0xF_FFFF	

3.2.4.1 Conditions and Behavior for Access to drseg, EJTAG Registers

The behavior of CPU access to the drseg address range at 0xFF30_0000 to 0xFF3F_FFFF is determined as shown in Table 3-4

Table 3-4 CPU Access to drseg Address Range

Transaction	LSNM bit in Debug register	Access
Load / Store	1	Kernel mode address space (kseg3)
Fetch	Don't care	drseg, see comments below
Load / Store	0	

Debug software is expected to read the debug control register (DCR) to determine which other memory mapped registers exist in drseg. The value returned in response to a read of any unimplemented memory mapped register is unpredictable, and writes are ignored to any unimplemented register in the drseg. Refer to Chapter 9, “EJTAG Debug Support,” for more information on the DCR.

The allowed access size is limited for the drseg. Only word size transactions are allowed. Operation of the processor is undefined for other transaction sizes.

3.2.4.2 Conditions and Behavior for Access to dmseg, EJTAG Memory

The behavior of CPU access to the dmseg address range at 0xFF20_0000 to 0xFF2F_FFFF is determined by the table shown in Table 3-5

Table 3-5 CPU Access to dmseg Address Range

Transaction	ProbEn bit in DCR register	LSNM bit in Debug register	Access
Load / Store	Don’t care	1	Kernel mode address space (kseg3)
Fetch	1	Don’t care	dmseg
Load / Store	1	0	
Fetch	0	Don’t care	See comments below
Load / Store	0	0	

The case with access to the dmseg when the ProbEn bit in the DCR register is 0 is not expected to happen. Debug software is expected to check the state of the ProbEn bit in DCR register before attempting to reference dmseg. If such a reference does happen, the reference hangs until it is satisfied by the probe. The probe can not assume that there will never be a reference to dmseg if the ProbEn bit in the DCR register is 0 because there is an inherent race between the debug software sampling the ProbEn bit as 1 and the probe clearing it to 0.

3.3 Translation Lookaside Buffer (4KEc Core Only)

The following subsections discuss the TLB memory management scheme used in the 4KEc processor core. The TLB consists of one joint and two micro address translation buffers:

- 16 dual-entry fully associative Joint TLB (JTLB)
- 4-entry fully associative Instruction micro TLB (ITLB)
- 4-entry fully associative Data micro TLB (DTLB)

3.3.1 Joint TLB

The 4KEc core implements a 16 dual-entry, fully associative Joint TLB that maps 32 virtual pages to their corresponding physical addresses. The purpose of the TLB is to translate virtual addresses and their corresponding ASID into a physical memory address. The translation is performed by comparing the upper bits of the virtual address (along with the ASID bits) against each of the entries in the *tag* portion of the JTLB structure. Because this structure is used to translate both instruction and data virtual addresses, it is referred to as a “joint” TLB.

The JTLB is organized as 16 pairs of even and odd entries containing descriptions of pages that range in size from 4-KBytes (or 1-KByte) to 256-MBytes into the 4-GByte physical address space. By default, the minimum page size is normally 4-KBytes on the 4KEc core; as a build-time option, it is possible to specify a minimum page size of 1-KByte.

The JTLB is organized in pairs of page entries to minimize its overall size. Each virtual *tag* entry corresponds to two physical data entries, an even page entry and an odd page entry. The highest order virtual address bit not participating in the tag comparison is used to determine which of the two data entries is used. Since page size can vary on a page-pair basis, the determination of which address bits participate in the comparison and which bit is used to make the even-odd selection must be done dynamically during the TLB lookup.

Figure 3-7 on page 43 shows the contents of one of the 16 dual-entries in the JTLB. The bit range indication in the figure serves to clarify which address bits are (or may be) affected during the translation process.

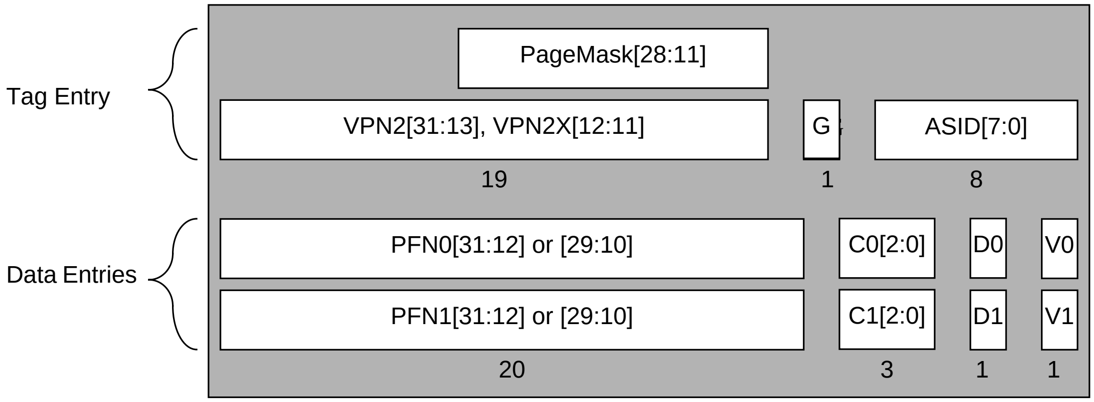


Figure 3-7 JTLB Entry (Tag and Data)

Table 3-6 and Table 3-7 explain each of the fields in a JTLB entry.

Table 3-6 TLB Tag Entry Fields

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td>PageMask[28:11]</td><td>Page Mask Value. The Page Mask defines the page size by masking the appropriate VPN2 bits from being involved in a comparison. It is also used to determine which address bit is used to make the even-odd page (PFN0-PFN1) determination. See the table below.
<table>
<tr><th>PageMask</th><th>Page Size</th><th>Even/Odd Bank Select Bit</th></tr>
<tr><td>00_0000_0000_0000_0000</td><td>1KB</td><td>VAddr[10]</td></tr>
<tr><td>00_0000_0000_0000_0011</td><td>4KB</td><td>VAddr[12]</td></tr>
<tr><td>00_0000_0000_0000_1111</td><td>16KB</td><td>VAddr[14]</td></tr>
<tr><td>00_0000_0000_0011_1111</td><td>64KB</td><td>VAddr[16]</td></tr>
<tr><td>00_0000_0000_1111_1111</td><td>256KB</td><td>VAddr[18]</td></tr>
<tr><td>00_0000_0011_1111_1111</td><td>1MB</td><td>VAddr[20]</td></tr>
<tr><td>00_0000_1111_1111_1111</td><td>4MB</td><td>VAddr[22]</td></tr>
<tr><td>00_0011_1111_1111_1111</td><td>16MB</td><td>VAddr[24]</td></tr>
<tr><td>00_1111_1111_1111_1111</td><td>64MB</td><td>VAddr[26]</td></tr>
<tr><td>11_1111_1111_1111_1111</td><td>256MB</td><td>VAddr[28]</td></tr>
</table>
The PageMask column above shows all the legal values for PageMask. Because each pair of bits can only have the same value, the physical entry in the JTLB will only save a compressed version of the PageMask using only 8 bits. This is however transparent to software, which will always work with a 18 bit field</td></tr>
</table>

Table 3-6 TLB Tag Entry Fields (Continued)

Field Name	Description
VPN2[31:13]	Virtual Page Number divided by 2. This field contains the upper bits of the virtual page number. Because it represents a pair of TLB pages, it is divided by 2. Bits 31:25 are always included in the TLB lookup comparison. Bits 24:13 are included depending on the page size, defined by PageMask.
VPN2X[12:11]	Extension to the VPN2 field to support 1KB pages.
G	Global Bit. When set, indicates that this entry is global to all processes and/or threads and thus disables inclusion of the ASID in the comparison.
ASID[7:0]	Address Space Identifier. Identifies which process or thread this TLB entry is associated with.

Table 3-7 TLB Data Entry Fields

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td>PFN0([31:12] or [29:10]),
PFN1([31:12] or [29:10])</td><td>Physical Frame Number. Defines the upper bits of the physical address.

The [29:10] range illustrates that if 1Kbytes page granularity is enabled, the PFN is shifted to the right, before being appended to the untranslated part of the virtual address. In this mode the upper two physical address bits are not covered by PFN but forced to zero.

For page sizes larger than 4 KBytes, only a subset of these bits is actually used.</td></tr>
<tr><td>C0[2:0],
C1[2:0]</td><td>Cacheability. Contains an encoded value of the cacheability attributes and determines whether the page should be placed in the cache or not. The field is encoded as follows:
<table>
<tr><th>C[2:0]</th><th>Coherency Attribute</th></tr>
<tr><td>000</td><td>Cacheable, noncoherent, write-through, no write-allocate</td></tr>
<tr><td>001</td><td>Cacheable, noncoherent, write-through, write-allocate</td></tr>
<tr><td>010</td><td>Uncached</td></tr>
<tr><td>011</td><td>Cacheable, noncoherent, write-back, write-allocate</td></tr>
<tr><td>100</td><td>Maps to entry 011b*</td></tr>
<tr><td>101</td><td>Maps to entry 011b*</td></tr>
<tr><td>110</td><td>Maps to entry 011b*</td></tr>
<tr><td>111</td><td>Maps to entry 010b*</td></tr>
<tr><td colspan="2">Note: * These mappings are not used on the 4KE processor cores but do have meaning in other MIPS Technologies implementations. Refer to the MIPS32 specification for more information.</td></tr>
</table>
</td></tr>
<tr><td>D0,
D1</td><td>“Dirty” or Write-enable Bit. Indicates that the page has been written and/or is writable. If this bit is set, stores to the page are permitted. If the bit is cleared, stores to the page cause a TLB Modified exception.</td></tr>
<tr><td>V0,
V1</td><td>Valid Bit. Indicates that the TLB entry and, thus, the virtual page mapping are valid. If this bit is set, accesses to the page are permitted. If the bit is cleared, accesses to the page cause a TLB Invalid exception.</td></tr>
</table>

In order to fill an entry in the JTLB, software executes a TLBWI or TLBWR instruction (See Section 3.4.3, "TLB Instructions" on page 49). Prior to invoking one of these instructions, several CP0 registers must be updated with the information to be written to a TLB entry:

- PageMask is set in the CP0 *PageMask* register.
- VPN2, VPN2X, and ASID are set in the CP0 *EntryHi* register.
- PFN0, C0, D0, V0 and G bits are set in the CP0 *EntryLo0* register.
- PFN1, C1, D1, V1 and G bits are set in the CP0 *EntryLo1* register.

Note that the global bit “G” is part of both *EntryLo0* and *EntryLo1*. The resulting “G” bit in the JTLB entry is the logical AND between the two fields in *EntryLo0* and *EntryLo1*. Please refer to Chapter 5, “CP0 Registers,” for further details.

The address space identifier (ASID) helps to reduce the frequency of TLB flushing on a context switch. The existence of the ASID allows multiple processes to exist in both the TLB and instruction caches. The ASID value is stored in the *EntryHi* register and is compared to the ASID value of each entry.

3.3.2 Instruction TLB

The ITLB is a small 4-entry, fully associative TLB dedicated to perform translations for the instruction stream. The ITLB only maps 4-Kbyte pages/sub-pages or 1-Kbyte pages/sub-pages if $Config3_{SP}$=1 and $PageGrain_{ESP}$=1.

The ITLB is managed by hardware and is transparent to software. If a fetch address cannot be translated by the ITLB, the JTLB is accessed trying to translate it in the following clock cycle. If successful, the translation information is copied into the ITLB. The ITLB is then re-accessed and the address will be successfully translated. This results in an ITLB miss penalty of at least 2 cycles. If the JTLB is busy with other operations, it may take additional cycles.

3.3.3 Data TLB

The DTLB is a small 4-entry, fully associative TLB which provides a faster translation for Load/Store addresses than is possible with the JTLB. The DTLB only maps 4-Kbyte pages/sub-pages or 1-Kbyte pages/sub-pages if $Config3_{SP}$=1 and $PageGrain_{ESP}$=1.

Like the ITLB, the DTLB is managed by hardware and is transparent to software. Unlike the ITLB, an access to the DTLB starts a parallel access to the JTLB. If there is a DTLB miss and a JTLB hit, the DTLB can be reloaded that cycle. The DTLB is then re-accessed and the translation will be successful. This parallel access reduces the DTLB miss penalty to 1 cycle.

3.4 Virtual-to-Physical Address Translation (4KEc Core)

Converting a virtual address to a physical address begins by comparing the virtual address from the processor with the virtual addresses in the TLB. There is a match when the VPN of the address is the same as the VPN field of the entry, and either:

- The Global (G) bit of both the even and odd pages of the TLB entry are set, or
- The ASID field of the virtual address is the same as the ASID field of the TLB entry

This match is referred to as a TLB *hit*. If there is no match, a TLB *miss* exception is taken by the processor and software is allowed to refill the TLB from a page table of virtual/physical addresses in memory.

Figure 3-8 on page 46 shows the logical translation of a virtual address into a physical address.

In this figure the virtual address is extended with an 8-bit ASID, which reduces the frequency of TLB flushing during a context switch. This 8-bit ASID contains the number assigned to that process and is stored in the CP0 *EntryHi* register.

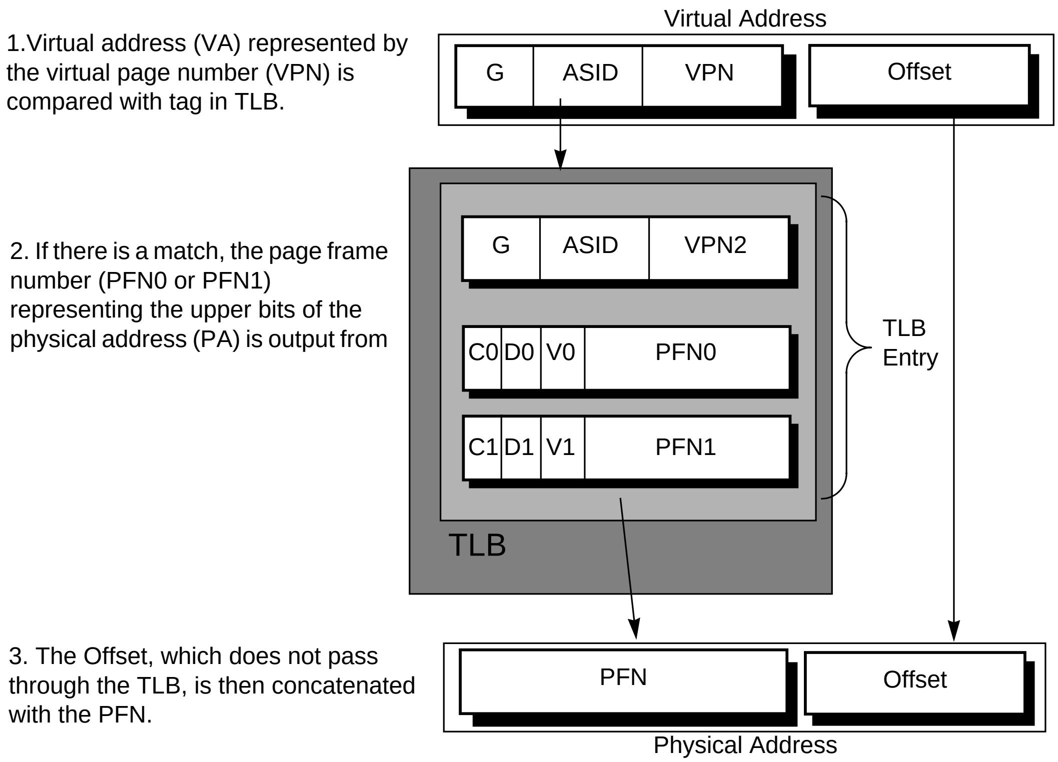


Figure 3-8 Overview of a Virtual-to-Physical Address Translation in the 4KEc Core

If there is a virtual address match in the TLB, the Physical Frame Number (PFN) is output from the TLB and concatenated with the *Offset*, to form the physical address. The *Offset* represents an address within the page frame space. As shown in Figure 3-8 on page 46, the *Offset* does not pass through the TLB. Figure 3-9 on page 47 shows a flow diagram of the 4KEc core address translation process for two page sizes. The top portion of the figure shows a virtual address for a 4 KByte page size. The width of the *Offset* is defined by the page size. The remaining 20 bits of the address represent the virtual page number (VPN). The bottom portion of Figure 3-9 on page 47 shows the virtual address for a 16 MByte page size. The remaining 8 bits of the address represent the VPN.

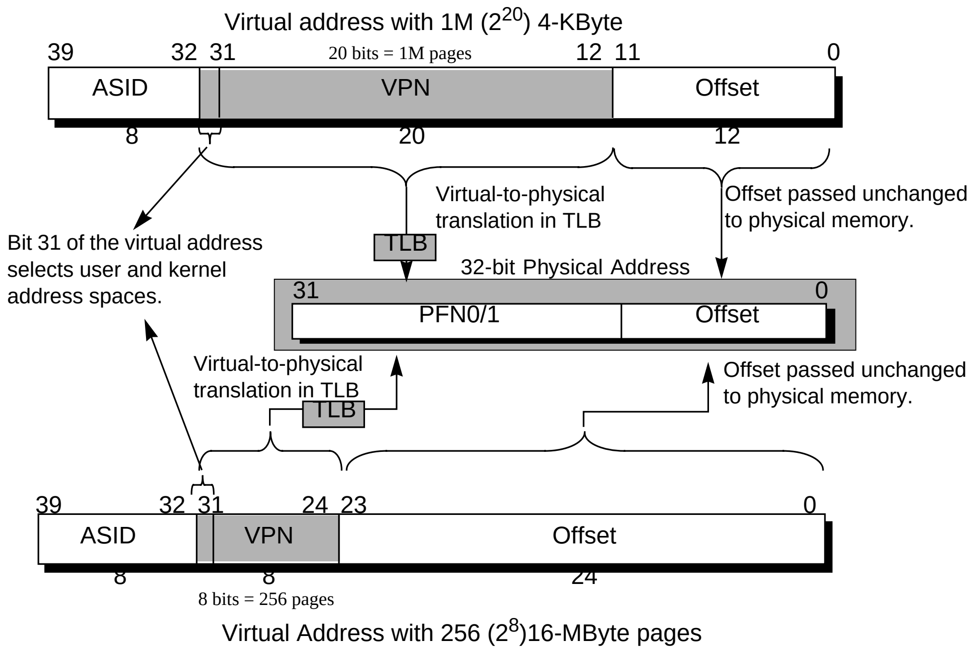


Figure 3-9 32-bit Virtual Address Translation

3.4.1 Hits, Misses, and Multiple Matches

Each JTLB entry contains a tag and two data fields. If a match is found, the upper bits of the virtual address are replaced with the page frame number (PFN) stored in the corresponding entry in the data array of the JTLB. The granularity of JTLB mappings is defined in terms of TLB pages. The 4KEc core JTLB supports pages of different sizes ranging from 1 KB to 256 MB in powers of 4. If a match is found, but the entry is invalid (i.e., the V bit in the data field is 0), a TLB Invalid exception is taken.If no match occurs (TLB miss), an exception is taken and software refills the TLB from the page table resident in memory. Figure 3-10 on page 49 shows the translation and exception flow of the TLB.

Software can write over a selected TLB entry or use a hardware mechanism to write into a random entry. The *Random* register selects which TLB entry to use on a TLBWR. This register decrements almost every cycle, wrapping to the maximum once its value is equal to the *Wired* register. Thus, TLB entries below the *Wired* value cannot be replaced by a TLBWR allowing important mappings to be preserved. In order to reduce the possibility for a livelock situation, the *Random* register includes a 10-bit LFSR that introduces a pseudo-random perturbation into the decrement.

The 4KEc core implements a TLB write-compare mechanism to ensure that multiple TLB matches do not occur. On the TLB write operation, the VPN2 field to be written is compared with all other entries in the TLB. If a match occurs, the 4KEc core takes a machine-check exception, sets the TS bit in the CP0 *Status* register, and aborts the write operation. For further details on exceptions, please refer to Chapter 4, “Exceptions and Interrupts,” on page 53. There is a hidden bit in each TLB entry that is cleared on a ColdReset. This bit is set once the TLB entry is written and is included in the match detection. Therefore, uninitialized TLB entries will not cause a TLB shutdown.

Note: This hidden initialization bit leaves the entire JTLB invalid after a ColdReset, eliminating the need to flush the TLB. But, to be compatible with other MIPS processors, it is recommended that software initialize all TLB entries with unique tag values and V bits cleared before the first access to a mapped location.

3.4.2 Memory Space

To assist in controlling both the amount of mapped space and the replacement characteristics of various memory regions, the 4KEc core provides two mechanisms.

3.4.2.1 Page Sizes

First, the page size can be configured, on a per entry basis, to map different page sizes ranging from 4 KByte to 256 MByte, in multiples of 4 (optionally, the 4KEc core can also support a smaller page size of 1 KByte). The CP0 *PageMask* register is loaded with the desired page size, which is then entered into the TLB when a new entry is written. Thus, operating systems can provide special-purpose maps. For example, a typical frame buffer can be memory mapped with only one TLB entry.

The 4KEc core implements the following page sizes:

(optionally 1K), 4K, 16K, 64K, 256K, 1M, 4M, 16M, 64M, 256M.

Software may determine which page sizes are supported by writing all ones to the CP0 *PageMask* register, then reading the value back. For additional information, see Section 5.2.5, "PageMask Register (CP0 Register 5, Select 0)" on page 96.

To enable support of 1 KByte pages in the 4KEc core a few steps must be taken. First, check that small pages are implemented by reading the CP0 $Config_{SP}$ bit. If set, small page sizes can be enabled by setting the ESP bit of the CP0 *PageGrain* register. See Section 5.2.6, "PageGrain Register (CP0 Register 5, Select 1)" on page 98 for more information.

3.4.2.2 Replacement Algorithm

The second mechanism controls the replacement algorithm when a TLB miss occurs. To select a TLB entry to be written with a new mapping, the 4KEc core provides a random replacement algorithm. However, the processor also provides a mechanism whereby a programmable number of mappings can be locked into the TLB via the CP0 Wired register, thus avoiding random replacement. Please refer to Section 5.2.7, "Wired Register (CP0 Register 6, Select 0)" on page 99 for further details.

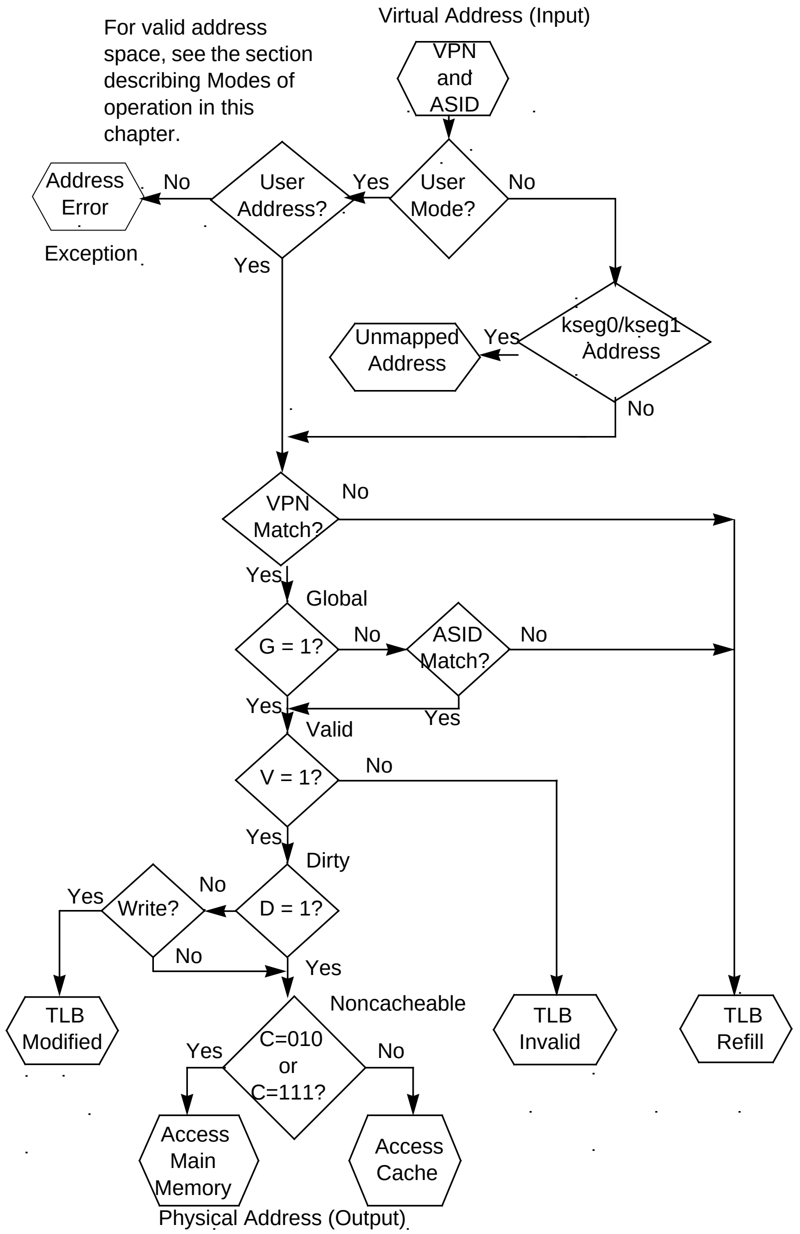


Figure 3-10 TLB Address Translation Flow in the 4KE Processor Core

3.4.3 TLB Instructions

Table 3-8 lists the 4KEc core's TLB-related instructions. Refer to Chapter 11, "MIPS32™ 4KE™ Processor Core Instructions," on page 237 for more information on these instructions.

Table 3-8 TLB Instructions

Op Code	Description of Instruction
TLBP	Translation Lookaside Buffer Probe
TLBR	Translation Lookaside Buffer Read
TLBWI	Translation Lookaside Buffer Write Index

Table 3-8 TLB Instructions

Op Code	Description of Instruction
TLBWR	Translation Lookaside Buffer Write Random

3.5 Fixed Mapping MMU (4KEm & 4KEp Cores)

The 4KEm and 4KEp cores implement a simple Fixed Mapping (FM) memory management unit that is smaller than the a full translation lookaside buffer (TLB) and more easily synthesized. Like a TLB, the FM performs virtual-to-physical address translation and provides attributes for the different memory segments. Those memory segments which are unmapped in a TLB implementation (kseg0 and kseg1) are translated identically by the FM in the 4KEm and 4KEp MMU.

The FM also determines the cacheability of each segment. These attributes are controlled via bits in the *Config* register. Table 3-9 shows the encoding for the K23 (bits 30:28), KU (bits 27:25) and K0 (bits 2:0) of the *Config* register.

Table 3-9 Cache Coherency Attributes

Config Register Fields K23, KU, and K0	Cache Coherency Attribute
0	Cacheable, noncoherent, write-through, no write-allocate
1	Cacheable, noncoherent, write-through, write-allocate
3, 4, 5, 6	Cacheable, noncoherent, write-back, write-allocate
2, 7	Uncached

In the 4KEm and 4KEp cores, no translation exceptions can be taken, although address errors are still possible.

Table 3-10 Cacheability of Segments with Block Address Translation

Segment	Virtual Address Range	Cacheability
useg/kuseg	0x0000_0000- 0x7FFF_FFFF	Controlled by the KU field (bits 27:25) of the *Config* register. Refer to Table 3-9 for the encoding.
kseg0	0x8000_0000- 0x9FFF_FFFF	Controlled by the K0 field (bits 2:0) of the *Config* register. See Table 3-9 for the encoding.
kseg1	0xA000_0000- 0xBFFF_FFFF	Always uncacheable
kseg2	0xC000_0000- 0xDFFF_FFFF	Controlled by the K23 field (bits 30:28) of the *Config* register. Refer to Table 3-9 for the encoding.
kseg3	0xE000_0000- 0xFFFF_FFFF	Controlled by K23 field (bits 30:28) of the *Config* register. Refer to Table 3-9 for the encoding.

The FM performs a simple translation to map from virtual addresses to physical addresses. This mapping is shown in Figure 3-11 on page 51. When ERL=1, useg and kuseg become unmapped and uncached. The ERL behavior is the same as if there was a TLB. The ERL mapping is shown in Figure 3-12 on page 52.

The ERL bit is usually never asserted by software. It is asserted by hardware after a Reset, SoftReset or NMI. See Section 4.8, "Exceptions" on page 69 for further information on exceptions.

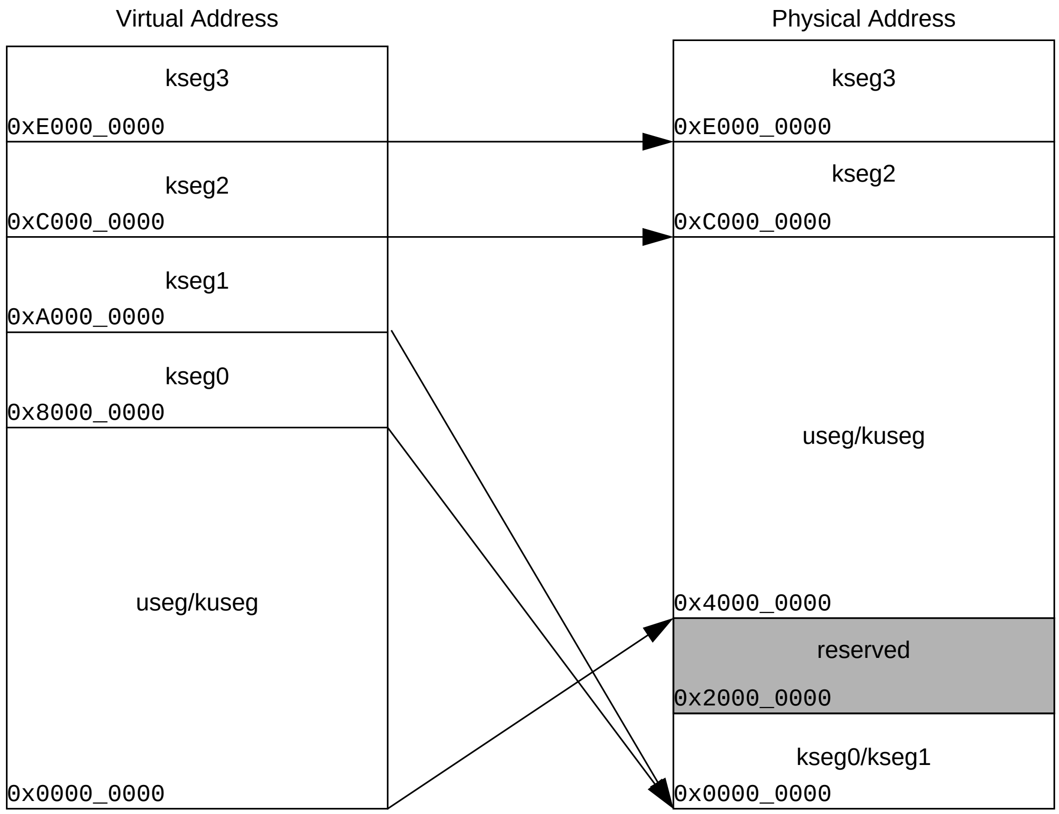


Figure 3-11 FM Memory Map (ERL=0) in the 4KEm and 4KEp Processor Cores

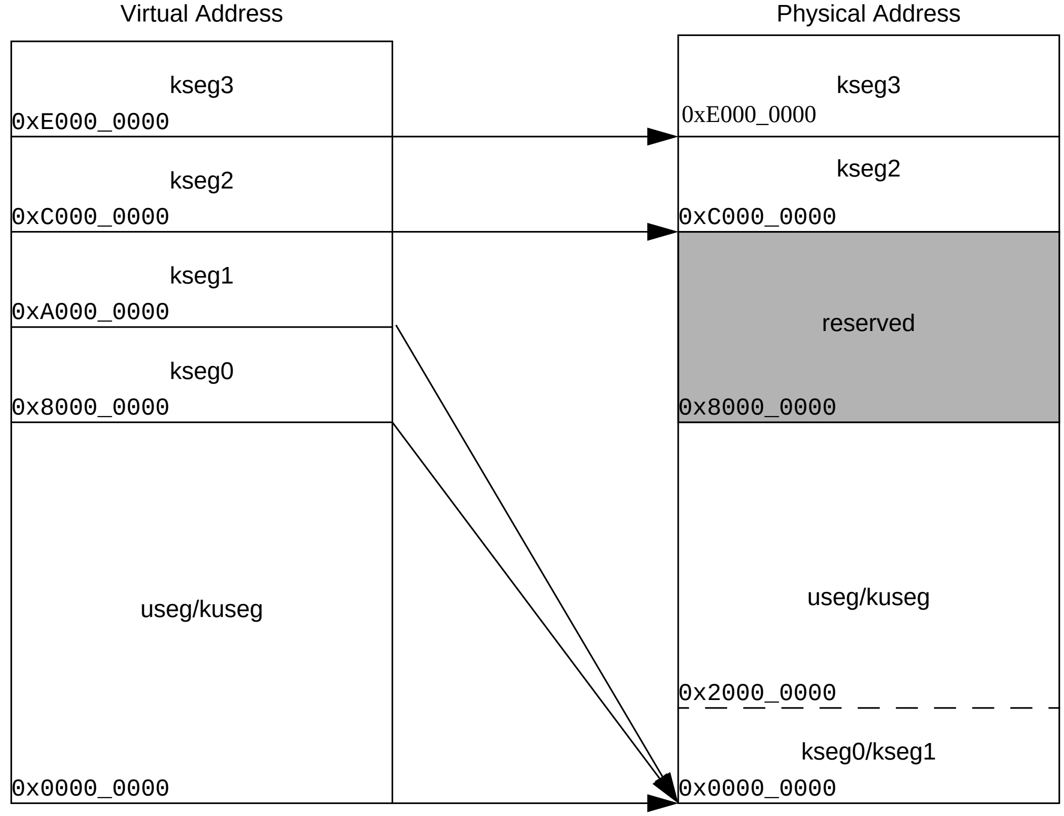


Figure 3-12 FM Memory Map (ERL=1) in the 4KEm and 4KEp Processor Cores

3.6 System Control Coprocessor

The System Control Coprocessor (CP0) is implemented as an integral part of the 4KE processor cores and supports memory management, address translation, exception handling, and other privileged operations. Certain CP0 registers are used to support memory management. Refer to Chapter 5, “CP0 Registers,” on page 87 for more information on the CP0 register set.

Exceptions and Interrupts

The MIPS32™ 4KE™ processor core receives exceptions from a number of sources, including translation lookaside buffer (TLB) misses, arithmetic overflows, I/O interrupts, and system calls. When the CPU detects one of these exceptions, the normal sequence of instruction execution is suspended and the processor enters kernel mode.

In kernel mode the core disables interrupts and forces execution of a software exception processor (called a handler) located at a specific address. The handler saves the context of the processor, including the contents of the program counter, the current operating mode, and the status of the interrupts (enabled or disabled). This context is saved so it can be restored when the exception has been serviced.

When an exception occurs, the core loads the *Exception Program Counter* (*EPC*) register with a location where execution can restart after the exception has been serviced. Most exceptions are *precise*, which mean that *EPC* can be used to identify the instruction that caused the exception. For precise exceptions the restart location in the *EPC* register is the address of the instruction that caused the exception or, if the instruction was executing in a branch delay slot, the address of the branch instruction immediately preceding the delay slot. To distinguish between the two, software must read the BD bit in the CP0 *Cause* register. Bus error exceptions and CP2 exceptions may be imprecise. For imprecise exceptions the instruction that caused the exception can not be identified.

This chapter contains the following sections:

4.1 Exception Conditions

When an exception condition occurs, the relevant instruction and all those that follow it in the pipeline are cancelled. Accordingly, any stall conditions and any later exception conditions that may have referenced this instruction are inhibited; there is no benefit in servicing stalls for a cancelled instruction.

When an exception condition is detected on an instruction fetch, the core aborts that instruction and all instructions that follow. When this instruction reaches the W stage, the exception flag causes it to write various CP0 registers with the exception state, change the current program counter (PC) to the appropriate exception vector address, and clear the exception bits of earlier pipeline stages.

This implementation allows all preceding instructions to complete execution and prevents all subsequent instructions from completing. Thus, the value in the *EPC* (*ErrorEPC* for errors, or *DEPC* for debug exceptions) is sufficient to restart

execution. It also ensures that exceptions are taken in the order of execution; an instruction taking an exception may itself be killed by an instruction further down the pipeline that takes an exception in a later cycle.

4.2 Exception Priority

Table 4-1 lists all possible exceptions, and the relative priority of each, highest to lowest. Several of these exceptions can happen simultaneously, in that event the exception with the highest priority is the one taken.

Table 4-1 Priority of Exceptions

Exception	Description
Reset	Assertion of SI_ColdReset signal.
Soft Reset	Assertion of SI_Reset signal.
DSS	EJTAG Debug Single Step.
DINT	EJTAG Debug Interrupt. Caused by the assertion of the external EJ_DINT input, or by setting the EjtagBrk bit in the *ECR* register.
NMI	Asserting edge of SI_NMI signal.
Machine Check	TLB write that conflicts with an existing entry (4KEc core).
Interrupt	Assertion of unmasked hardware or software interrupt signal.
Deferred Watch	Deferred Watch (unmasked by K|DM->!(K|DM) transition).
DIB	EJTAG debug hardware instruction break matched.
WATCH	A reference to an address in one of the watch registers (fetch).
AdEL	Fetch address alignment error. User mode fetch reference to kernel address.
TLBL	Fetch TLB miss (4KEc core). Fetch TLB hit to page with V=0 (4KEc core).
IBE	Instruction fetch bus error.
DBp	EJTAG Breakpoint (execution of SDBBP instruction).
Sys	Execution of SYSCALL instruction.
Bp	Execution of BREAK instruction.
CpU	Execution of a coprocessor instruction for a coprocessor that is not enabled.
RI	Execution of a Reserved Instruction.
C2E	Execution of coprocessor 2 instruction which caused a general exception in the coprocessor.
IS1	Execution of coprocessor 2 instruction which caused an Implementation Specific exception 1 in the coprocessor.
IS2	Execution of coprocessor 2 instruction which caused an Implementation Specific exception 2 in the coprocessor.
Ov	Execution of an arithmetic instruction that overflowed.
Tr	Execution of a trap (when trap condition is true).

Table 4-1 Priority of Exceptions (Continued)

Exception	Description
DDBL / DDBS	EJTAG Data Address Break (address only) or EJTAG Data Value Break on Store (address and value).
WATCH	A reference to an address in one of the watch registers (data).
AdEL	Load address alignment error. User mode load reference to kernel address.
AdES	Store address alignment error. User mode store to kernel address.
TLBL	Load TLB miss (4KEc core). Load TLB hit to page with V=0 (4KEc core).
TLBS	Store TLB miss (4KEc core). Store TLB hit to page with V=0 (4KEc core).
TLB Mod	Store to TLB page with D=0 (4KEc core).
DBE	Load or store bus error.
DDBL	EJTAG data hardware breakpoint matched in load data compare.

4.3 Interrupts

Older 32-bit cores available from MIPS that implemented Release 1 of the Architecture included support for two software interrupts, six hardware interrupts, and a special-purpose timer interrupt. (Note that the Architecture also defines a performance counter interrupt, but this is not implemented on the 4KE core.) The timer interrupt was provided external to the core and typically combined with hardware interrupt 5 in an system-dependent manner. Interrupts were handled either through the general exception vector (offset 16#180) or the special interrupt vector (16#200), based on the value of $\mathrm{Cause}_{\mathrm{IV}}$. Software was required to prioritize interrupts as a function of the $\mathrm{Cause}_{\mathrm{IP}}$ bits in the interrupt handler prologue.

Release 2 of the Architecture, implemented by the 4KE core, adds an upward-compatible extension to the Release 1 interrupt architecture that supports vectored interrupts. In addition, Release 2 adds a new interrupt mode that supports the use of an external interrupt controller by changing the interrupt architecture.

4.3.1 Interrupt Modes

The 4KE core includes support for three interrupt modes, as defined by Release 2 of the Architecture:

- Interrupt compatibility mode, which acts identically to that in an implementation of Release 1 of the Architecture.
- Vectored Interrupt (VI) mode, which adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt, and to assign a GPR shadow set for use during interrupt processing. The presence of this mode is denoted by the VInt bit in the *Config3* register. This mode is architecturally optional; but it is always present on the 4KE core, so the VInt bit will always read as a 1 for the 4KE core.
- External Interrupt Controller (EIC) mode, which redefines the way in which interrupts are handled to provide full support for an external interrupt controller handling prioritization and vectoring of interrupts. This presence of this mode denoted by the VEIC bit in the *Config3* register. Again, this mode is architecturally optional. On the 4KE core,

the VEIC bit is set externally by the static input, *SI_EICPresent*, to allow system logic to indicate the presence of an external interrupt controller.

The reset state of the processor is to interrupt compatibility mode such that a processor supporting Release 2 of the Architecture, like the 4KE core, is fully compatible with implementations of Release 1 of the Architecture.

Table 4-2 shows the current interrupt mode of the processor as a function of the coprocessor 0 register fields that can affect the mode.

Table 4-2 Interrupt Modes

$Status_{BEV}$	$Cause_{IV}$	$IntCtl_{VS}$	$Config3_{VINT}$	$Config3_{VEIC}$	Interrupt Mode
1	x	x	x	x	Compatibly
x	0	x	x	x	Compatibility
x	x	=0	x	x	Compatibility
0	1	≠0	1	0	Vectored Interrupt
0	1	≠0	x	1	External Interrupt Controller
0	1	≠0	0	0	Can’t happen - $IntCtl_{VS}$ can not be non-zero if neither Vectored Interrupt nor External Interrupt Controller mode is implemented.
“x” denotes don’t care					

4.3.1.1 Interrupt Compatibility Mode

This is the default interrupt mode for the processor and is entered when a Reset exception occurs. In this mode, interrupts are non-vectored and dispatched though exception vector offset 16#180 (if $Cause_{IV}$ = 0) or vector offset 16#200 (if $Cause_{IV}$ = 1). This mode is in effect if any of the following conditions are true:

- $Cause_{IV}$ = 0
- $Status_{BEV}$ = 1
- $IntCtl_{VS}$ = 0, which would be the case if vectored interrupts are not implemented, or have been disabled.

A typical software handler for interrupt compatibility mode might look as follows:

```
/*
 * Assumptions:
 *  - CauseIV = 1 (if it were zero, the interrupt exception would have to
 *                be isolated from the general exception vector before getting
 *                here)
 *  - GPRs k0 and k1 are available (no shadow register switches invoked in
 *                                  compatibility mode)
 *  - The software priority is IP7..IP0 (HW5..HW0, SW1..SW0)
 *
 * Location: Offset 0x200 from exception base
 */

IVexception:
   mfc0   k0, C0_Cause        /* Read Cause register for IP bits */
   mfc0   k1, C0_Status       /* and Status register for IM bits */
```

```
    andi   k0, k0, M_CauseIM   /* Keep only IP bits from Cause */
    and    k0, k0, k1          /* and mask with IM bits */
    beq    k0, zero, Dismiss   /* no bits set - spurious interrupt */
    clz    k0, k0              /* Find first bit set, IP7..IP0; k0 = 16..23 */
    xori   k0, k0, 0x17        /* 16..23 => 7..0 */
    sll    k0, k0, VS          /* Shift to emulate software IntCtlVS */
    la     k1, VectorBase      /* Get base of 8 interrupt vectors */
    addu   k0, k0, k1          /* Compute target from base and offset */
    jr     k0                  /* Jump to specific exception routine */
    nop

/*
 * Each interrupt processing routine processes a specific interrupt, analogous
 * to those reached in VI or EIC interrupt mode. Since each processing routine
 * is dedicated to a particular interrupt line, it has the context to know
 * which line was asserted.  Each processing routine may need to look further
 * to determine the actual source of the interrupt if multiple interrupt requests
 * are ORed together on a single IP line. Once that task is performed, the
 * interrupt may be processed in one of two ways:
 *
 * - Completely at interrupt level (e.g., a simply UART interrupt). The
 *   SimpleInterrupt routine below is an example of this type.
 * - By saving sufficient state and re-enabling other interrupts. In this
 *   case the software model determines which interrupts are disabled during
 *   the processing of this interrupt. Typically, this is either the single
 *   StatusIM bit that corresponds to the interrupt being processed, or some
 *   collection of other StatusIM bits so that “lower” priority interrupts are
 *    also disabled. The NestedInterrupt routine below is an example of this type.
 */

SimpleInterrupt:
/*
 * Process the device interrupt here and clear the interupt request
 * at the device. In order to do this, some registers may need to be
 * saved and restored. The coprocessor 0 state is such that an ERET
 * will simple return to the interrupted code.
 */
    eret                       /* Return to interrupted code */

NestedException:
/*
 * Nested exceptions typically require saving the EPC and Status registers,
 * any GPRs that may be modified by the nested exception routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

    /* Save GPRs here, and setup software context */
    mfc0   k0, C0_EPC          /* Get restart address */
    sw     k0, EPCSave         /* Save in memory */
    mfc0   k0, C0_Status       /* Get Status value */
    sw     k0, StatusSave      /* Save in memory */
    li     k1, ~IMbitsToClear  /* Get Im bits to clear for this interrupt */
                               /*   this must include at least the IM bit */
                               /*   for the current interrupt, and may include */
                               /*   others */
    and    k0, k0, k1             /* Clear bits in copy of Status */
    ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
```

```
                                        /* Clear KSU, ERL, EXL bits in k0 */
    mtc0    k0, C0_Status               /* Modify mask, switch to kernel mode, */
                                        /*   re-enable interrupts */

    /*
     * Process interrupt here, including clearing device interrupt.
     * In some environments this may be done with a thread running in
     * kernel or user mode. Such an environment is well beyond the scope of
     * this example.
     */

/*
 * To complete interrupt processing, the saved values must be restored
 * and the original interrupted code restarted.
 */

    di                              /* Disable interrupts - may not be required */
    lw      k0, StatusSave          /* Get saved Status (including EXL set) */
    lw      k1, EPCSave             /*   and EPC */
    mtc0    k0, C0_Status           /* Restore the original value */
    mtc0    k1, C0_EPC              /*  and EPC */
    /* Restore GPRs and software state */
    eret                            /* Dismiss the interrupt */
```

4.3.1.2 Vectored Interrupt Mode

Vectored Interrupt mode builds on the interrupt compatibility mode by adding a priority encoder to prioritize pending interrupts and to generate a vector with which each interrupt can be directed to a dedicated handler routine. This mode also allows each interrupt to be mapped to a GPR shadow set for use by the interrupt handler. Vectored Interrupt mode is in effect if all of the following conditions are true:

- $\text{Config3}_{\text{VInt}} = 1$
- $\text{Config3}_{\text{VEIC}} = 0$
- $\text{IntCtl}_{\text{VS}} \neq 0$
- $\text{Cause}_{\text{IV}} = 1$
- $\text{Status}_{\text{BEV}} = 0$

In VI interrupt mode, the six hardware interrupts are interpreted as individual hardware interrupt requests. The timer interrupt is combined in a system-dependent way (external to the core) with the hardware interrupts (the interrupt with which they are combined is indicated by the $IntCtl_{IPTI}$ field) to provide the appropriate relative priority of the timer interrupt with that of the hardware interrupts. The processor interrupt logic ANDs each of the $Cause_{IP}$ bits with the corresponding $Status_{IM}$ bits. If any of these values is 1, and if interrupts are enabled ($\text{Status}_{\text{IE}} = 1$, $\text{Status}_{\text{EXL}} = 0$, and $\text{Status}_{\text{ERL}} = 0$), an interrupt is signaled and a priority encoder scans the values in the order shown in Table 4-3.

Table 4-3 Relative Interrupt Priority for Vectored Interrupt Mode

Relative Priority	Interrupt Type	Interrupt Source	Interrupt Request Calculated From	Vector Number Generated by Priority Encoder
Highest Priority	Hardware	HW5	IP7 and IM7	7
		HW4	IP6 and IM6	6
		HW3	IP5 and IM5	5
		HW2	IP4 and IM4	4
		HW1	IP3 and IM3	3
		HW0	IP2 and IM2	2
	Software	SW1	IP1 and IM1	1
Lowest Priority		SW0	IP0 and IM0	0

The priority order places a relative priority on each hardware interrupt and places the software interrupts at a priority lower than all hardware interrupts. When the priority encoder finds the highest priority pending interrupt, it outputs an encoded vector number that is used in the calculation of the handler for that interrupt, as described below. This is shown pictorially in Figure 4-1.

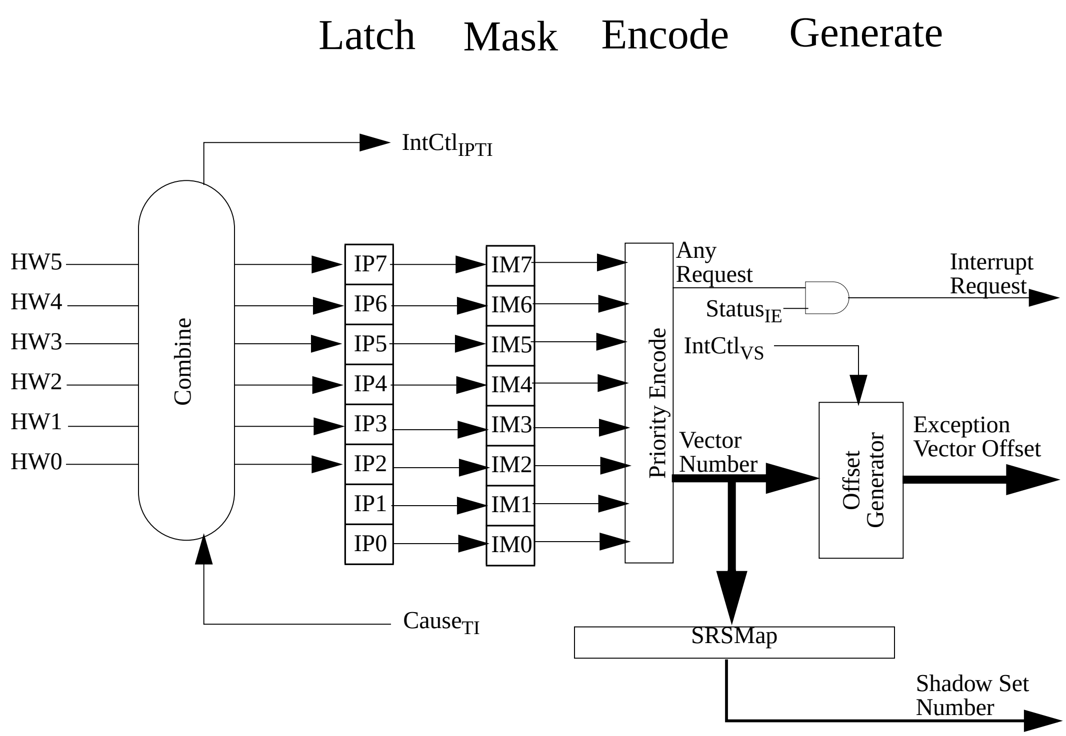


Figure 4-1 Interrupt Generation for Vectored Interrupt Mode

A typical software handler for vectored interrupt mode bypasses the entire sequence of code following the IVexception label shown for the compatibility mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine. Unlike the compatibility mode examples, a vectored interrupt handler may take advantage of a dedicated GPR shadow set to avoid saving any registers. As such, the SimpleInterrupt code shown above need not save the GPRs.

A nested interrupt is similar to that shown for compatibility mode, but may also take advantage of running the nested exception routine in the GPR shadow set dedicated to the interrupt or in another shadow set. Such a routine might look as follows:

```
NestedException:
/*
 * Nested exceptions typically require saving the EPC, Status and SRSCtl registers,
 * setting up the appropriate GPR shadow set for the routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

   /* Use the current GPR shadow set, and setup software context */
   mfc0   k0, C0_EPC          /* Get restart address */
   sw     k0, EPCSave         /* Save in memory */
   mfc0   k0, C0_Status       /* Get Status value */
   sw     k0, StatusSave      /* Save in memory */
   mfc0   k0, C0_SRSCtl       /* Save SRSCtl if changing shadow sets */
   sw     k0, SRSCtlSave
   li     k1, ~IMbitsToClear  /* Get Im bits to clear for this interrupt */
                              /*   this must include at least the IM bit */
                              /*   for the current interrupt, and may include */
                              /*   others */
   and    k0, k0, k1              /* Clear bits in copy of Status */
   /* If switching shadow sets, write new value to SRSCtlPSS here */
   ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                  /* Clear KSU, ERL, EXL bits in k0 */
   mtc0   k0, C0_Status           /* Modify mask, switch to kernel mode, */
                                  /*   re-enable interrupts */
   /*
    * If switching shadow sets, clear only KSU above, write target
    * address to EPC, and do execute an eret to clear EXL, switch
    * shadow sets, and jump to routine
    */

   /* Process interrupt here, including clearing device interrupt */

/*
 * To complete interrupt processing, the saved values must be restored
 * and the original interrupted code restarted.
 */

   di                         /* Disable interrupts - may not be required */
   lw     k0, StatusSave      /* Get saved Status (including EXL set) */
   lw     k1, EPCSave         /*   and EPC */
   mtc0   k0, C0_Status       /* Restore the original value */
   lw     k0, SRSCtlSave      /* Get saved SRSCtl */
   mtc0   k1, C0_EPC          /*  and EPC */
   mtc0   k0, C0_SRSCtl       /* Restore shadow sets */
   ehb                        /* Clear hazard */
   eret                       /* Dismiss the interrupt */
```

4.3.1.3 External Interrupt Controller Mode

External Internal Interrupt Controller Mode redefines the way that the processor interrupt logic is configured to provide support for an external interrupt controller. The interrupt controller is responsible for prioritizing all interrupts, including

hardware, software, timer, and performance counter interrupts, and directly supplying to the processor the vector number of the highest priority interrupt. EIC interrupt mode is in effect if all of the following conditions are true:

- $\text{Config3}_{\text{VEIC}} = 1$
- $\text{IntCtl}_{\text{VS}} \neq 0$
- $\text{Cause}_{\text{IV}} = 1$
- $\text{Status}_{\text{BEV}} = 0$

In EIC interrupt mode, the processor sends the state of the software interrupt requests ($\text{Cause}_{\text{IP1..IP0}}$) and the timer interrupt request (Cause_{TI}) to the external interrupt controller, where it prioritizes these interrupts in a system-dependent way with other hardware interrupts. The interrupt controller can be a hard-wired logic block, or it can be configurable based on control and status registers. This allows the interrupt controller to be more specific or more general as a function of the system environment and needs.

The external interrupt controller prioritizes its interrupt requests and produces the vector number of the highest priority interrupt to be serviced. The vector number, called the Requested Interrupt Priority Level (RIPL), is a 6-bit encoded value in the range 0..63, inclusive. A value of 0 indicates that no interrupt requests are pending. The values 1..63 represent the lowest (1) to highest (63) RIPL for the interrupt to be serviced. The interrupt controller passes this value on the 6 hardware interrupt line, which are treated as an encoded value in EIC interrupt mode.

$\text{Status}_{\text{IPL}}$ (which overlays $\text{Status}_{\text{IM7..IM2}}$) is interpreted as the Interrupt Priority Level (IPL) at which the processor is currently operating (with a value of zero indicating that no interrupt is currently being serviced). When the interrupt controller requests service for an interrupt, the processor compares RIPL with $\text{Status}_{\text{IPL}}$ to determine if the requested interrupt has higher priority than the current IPL. If RIPL is strictly greater than $\text{Status}_{\text{IPL}}$, and interrupts are enabled ($\text{Status}_{\text{IE}} = 1$, $\text{Status}_{\text{EXL}} = 0$, and $\text{Status}_{\text{ERL}} = 0$) an interrupt request is signaled to the pipeline. When the processor starts the interrupt exception, it loads RIPL into $\text{Cause}_{\text{RIPL}}$ (which overlays $\text{Cause}_{\text{IP7..IP2}}$) and signals the external interrupt controller to notify it that the request is being serviced. The interrupt exception uses the value of $\text{Cause}_{\text{RIPL}}$ as the vector number. Because $\text{Cause}_{\text{RIPL}}$ is only loaded by the processor when an interrupt exception is signaled, it is available to software during interrupt processing.

In EIC interrupt mode, the external interrupt controller is also responsible for supplying the GPR shadow set number to use when servicing the interrupt. As such, the *SRSMap* register is not used in this mode, and the mapping of the vectored interrupt to a GPR shadow set is done by programming (or designing) the interrupt controller to provide the correct GPR shadow set number when an interrupt is requested. When the processor loads an interrupt request into $\text{Cause}_{\text{RIPL}}$, it also loads the GPR shadow set number into $\text{SRSCtl}_{\text{EICSS}}$, which is copied to $\text{SRSCtl}_{\text{CSS}}$ when the interrupt is serviced.

The operation of EIC interrupt mode is shown pictorially in Figure 4-2.

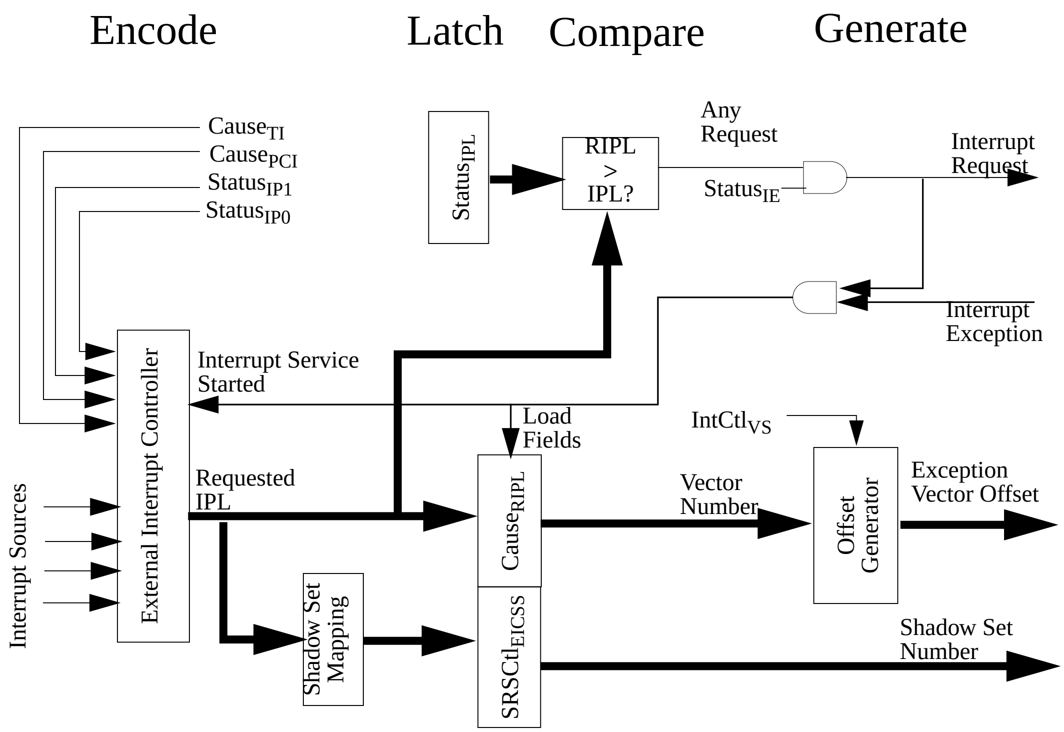


Figure 4-2 Interrupt Generation for External Interrupt Controller Interrupt Mode

A typical software handler for EIC interrupt mode bypasses the entire sequence of code following the IVexception label shown for the compatibility mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine. Unlike the compatibility mode examples, an EIC interrupt handler may take advantage of a dedicated GPR shadow set to avoid saving any registers. As such, the SimpleInterrupt code shown above need not save the GPRs.

A nested interrupt is similar to that shown for compatibility mode, but may also take advantage of running the nested exception routine in the GPR shadow set dedicated to the interrupt or in another shadow set. It also need only copy $\text{Cause}_{\text{RIPL}}$ to $\text{Status}_{\text{IPL}}$ to prevent lower priority interrupts from interrupting the handler. Such a routine might look as follows:

```
NestedException:
/*
 * Nested exceptions typically require saving the EPC, Status,and SRSCtl registers,
 * setting up the appropriate GPR shadow set for the routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

   /* Use the current GPR shadow set, and setup software context */
   mfc0   k1, C0_Cause        /* Read Cause to get RIPL value */
   mfc0   k0, C0_EPC          /* Get restart address */
   srl    k1, k1, S_CauseRIPL /* Right justify RIPL field */
   sw     k0, EPCSave         /* Save in memory */
   mfc0   k0, C0_Status       /* Get Status value */
   sw     k0, StatusSave      /* Save in memory */
   ins    k0, k1, S_StatusIPL, 6 /* Set IPL to RIPL in copy of Status */
   mfc0   k1, C0_SRSCtl       /* Save SRSCtl if changing shadow sets */
   sw     k1, SRSCtlSave
```

```
    /* If switching shadow sets, write new value to SRSCtlPSS here */
    ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                  /* Clear KSU, ERL, EXL bits in k0 */
    mtc0   k0, C0_Status          /* Modify IPL, switch to kernel mode, */
                                  /*   re-enable interrupts */
    /*
     * If switching shadow sets, clear only KSU above, write target
     * address to EPC, and do execute an eret to clear EXL, switch
     * shadow sets, and jump to routine
     */

    /* Process interrupt here, including clearing device interrupt */

/*
 * The interrupt completion code is identical to that shown for VI mode above.
 */
```

4.3.2 Generation of Exception Vector Offsets for Vectored Interrupts

For vectored interrupts (in either VI or EIC interrupt mode), a vector number is produced by the interrupt control logic. This number is combined with $IntCtl_{VS}$ to create the interrupt offset, which is added to 16#200 to create the exception vector offset. For VI interrupt mode, the vector number is in the range 0..7, inclusive. For EIC interrupt mode, the vector number is in the range 1..63, inclusive (0 being the encoding for “no interrupt”). The $IntCtl_{VS}$ field specifies the spacing between vector locations. If this value is zero (the default reset state), the vector spacing is zero and the processor reverts to Interrupt Compatibility Mode. A non-zero value enables vectored interrupts, and Table 4-4 shows the exception vector offset for a representative subset of the vector numbers and values of the $IntCtl_{VS}$ field.

Table 4-4 Exception Vector Offsets for Vectored Interrupts

Vector Number	Value of $IntCtl_{VS}$ Field				
	2#00001	2#00010	2#00100	2#01000	2#10000
0	16#0200	16#0200	16#0200	16#0200	16#0200
1	16#0220	16#0240	16#0280	16#0300	16#0400
2	16#0240	16#0280	16#0300	16#0400	16#0600
3	16#0260	16#02C0	16#0380	16#0500	16#0800
4	16#0280	16#0300	16#0400	16#0600	16#0A00
5	16#02A0	16#0340	16#0480	16#0700	16#0C00
6	16#02C0	16#0380	16#0500	16#0800	16#0E00
7	16#02E0	16#03C0	16#0580	16#0900	16#1000
• • •					
61	16#09A0	16#1140	16#2080	16#3F00	16#7C00
62	16#09C0	16#1180	16#2100	16#4000	16#7E00
63	16#09E0	16#11C0	16#2180	16#4100	16#8000

The general equation for the exception vector offset for a vectored interrupt is:

$$\texttt{vectorOffset} \leftarrow \texttt{16\#200 + (vectorNumber} \times \texttt{(IntCtl}_{VS} \parallel \texttt{2\#00000))}$$

4.4 GPR Shadow Registers

Release 2 of the Architecture optionally removes the need to save and restore GPRs on entry to high priority interrupts or exceptions, and to provide specified processor modes with the same capability. This is done by introducing multiple copies of the GPRs, called *shadow sets*, and allowing privileged software to associate a shadow set with entry to kernel mode via an interrupt vector or exception. The normal GPRs are logically considered shadow set zero.

The number of GPR shadow sets is a build-time option on the 4KE core. Although Release 2 of the Architecture defines a maximum of 16 shadow sets, the core allows one (the normal GPRs), two, or four shadow sets. The highest number actually implemented is indicated by the $SRSCtl_{HSS}$ field. If this field is zero, only the normal GPRs are implemented.

Shadow sets are new copies of the GPRs that can be substituted for the normal GPRs on entry to kernel mode via an interrupt or exception. Once a shadow set is bound to a kernel mode entry condition, reference to GPRs work exactly as one would expect, but they are redirected to registers that are dedicated to that condition. Privileged software may need to reference all GPRs in the register file, even specific shadow registers that are not visible in the current mode. The RDPGPR and WRPGPR instructions are used for this purpose. The CSS field of the *SRSCtl* register provides the number of the current shadow register set, and the PSS field of the *SRSCtl* register provides the number of the previous shadow register set (that which was current before the last exception or interrupt occurred).

If the processor is operating in VI interrupt mode, binding of a vectored interrupt to a shadow set is done by writing to the *SRSMap* register. If the processor is operating in EIC interrupt mode, the binding of the interrupt to a specific shadow set is provided by the external interrupt controller, and is configured in an implementation-dependent way. Binding of an exception or non-vectored interrupt to a shadow set is done by writing to the ESS field of the *SRSCtl* register. When an exception or interrupt occurs, the value of $SRSCtl_{CSS}$ is copied to $SRSCtl_{PSS}$, and $SRSCtl_{CSS}$ is set to the value taken from the appropriate source. On an ERET, the value of $SRSCtl_{PSS}$ is copied back into $SRSCtl_{CSS}$ to restore the shadow set of the mode to which control returns. More precisely, the rules for updating the fields in the *SRSCtl* register on an interrupt or exception are as follows:

1. No field in the *SRSCtl* register is updated if any of the following conditions is true. In this case, steps 2 and 3 are skipped.
   - The exception is one that sets $Status_{ERL}$: Reset, Soft Reset, or NMI.
   - The exception causes entry into EJTAG Debug Mode
   - $Status_{BEV} = 1$
   - $Status_{EXL} = 1$
2. $SRSCtl_{CSS}$ is copied to $SRSCtl_{PSS}$
3. $SRSCtl_{CSS}$ is updated from one of the following sources:
   - The appropriate field of the *SRSMap* register, based on IPL, if the exception is an interrupt, $Cause_{IV} = 1$, $Config3_{VEIC} = 0$, and $Config3_{VInt} = 1$. These are the conditions for a vectored interrupt.
   - The EICSS field of the *SRSCtl* register if the exception is an interrupt, $Cause_{IV} = 1$, and $Config3_{VEIC} = 1$. These are the conditions for a vectored EIC interrupt.
   - The ESS field of the *SRSCtl* register in any other case. This is the condition for a non-interrupt exception, or a non-vectored interrupt.

Similarly, the rules for updating the fields in the *SRSCtl* register at the end of an exception or interrupt are as follows:

1. No field in the *SRSCtl* register is updated if any of the following conditions is true. In this case, step 2 is skipped.
   - A DERET is executed
   - An ERET is executed with $Status_{ERL}$ = 1
2. $SRSCtl_{PSS}$ is copied to $SRSCtl_{CSS}$

These rules have the effect of preserving the *SRSCtl* register in any case of a nested exception or one which occurs before the processor has been fully initialize ($Status_{BEV}$ = 1).

Privileged software may switch the current shadow set by writing a new value into $SRSCtl_{PSS}$, loading EPC with a target address, and doing an ERET.

4.5 Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to location `16#BFC0.0000`. EJTAG Debug exceptions are vectored to location `16#BFC0.0480`, or to location `16#FF20.0200` if the ProbTrap bit is zero or one, respectively, in the EJTAG_Control_register. Addresses for all other exceptions are a combination of a vector offset and a vector base address. In Release 1 of the architecture, the vector base address was fixed. In Release 2 of the architecture, software is allowed to specify the vector base address via the *EBase* register for exceptions that occur when $Status_{BEV}$ equals 0. Table 4-5 gives the vector base address as a function of the exception and whether the BEV bit is set in the *Status* register. Table 4-6 gives the offsets from the vector base address as a function of the exception. Note that the IV bit in the *Cause* register causes Interrupts to use a dedicated exception vector offset, rather than the general exception vector. For implementations of Release 2 of the Architecture, Table 4-4 gives the offset from the base address in the case where $Status_{BEV}$ = 0 and $Cause_{IV}$ = 1. For implementations of Release 1 of the architecture in which $Cause_{IV}$ = 1, the vector offset is as if $IntCtl_{VS}$ were 0. Table 4-7 combines these two tables into one that contains all possible vector addresses as a function of the state that can affect the vector selection. To avoid complexity in the table, the vector address value assumes that the *EBase* register, as implemented in Release 2 devices, is not changed from its reset state and that $IntCtl_{VS}$ is 0.

Table 4-5 Exception Vector Base Addresses

Exception	$Status_{BEV}$ = 0	$Status_{BEV}$ = 1
Reset, Soft Reset, NMI	16#BFC0.0000	
EJTAG Debug (with ProbEn = 0 in the EJTAG_Control_register)	16#BFC0.0480	
EJTAG Debug (with ProbEn = 1 in the EJTAG_Control_register)	16#FF20.0200	
Other	*For Release 1 of the architecture:* 16#8000.0000 *For Release 2 of the architecture:* $EBase_{31..12}$ ‖ 16#000 Note that $EBase_{31..30}$ have the fixed value 2#10	16#BFC0.0200

Table 4-6 Exception Vector Offsets

Exception	Vector Offset
TLB Refill, EXL = 0	`16#000`
General Exception	`16#180`
Interrupt, $Cause_{IV}$ = 1	`16#200` (In Release 2 implementations, this is the base of the vectored interrupt table when $Status_{BEV}$ = 0)
Reset, Soft Reset, NMI	None (Uses Reset Base Address)

Table 4-7 Exception Vectors

Exception	$Status_{BEV}$	$Status_{EXL}$	$Cause_{IV}$	EJTAG ProbEn	Vector For Release 2 Implementations, assumes that EBase retains its reset state and that $IntCtl_{VS}$ = 0
Reset, Soft Reset, NMI	x	x	x	x	`16#BFC0.0000`
EJTAG Debug	x	x	x	0	`16#BFC0.0480`
EJTAG Debug	x	x	x	1	`16#FF20.0200`
TLB Refill	0	0	x	x	`16#8000.0000`
TLB Refill	0	1	x	x	`16#8000.0180`
TLB Refill	1	0	x	x	`16#BFC0.0200`
TLB Refill	1	1	x	x	`16#BFC0.0380`
Interrupt	0	0	0	x	`16#8000.0180`
Interrupt	0	0	1	x	`16#8000.0200`
Interrupt	1	0	0	x	`16#BFC0.0380`
Interrupt	1	0	1	x	`16#BFC0.0400`
All others	0	x	x	x	`16#8000.0180`
All others	1	x	x	x	`16#BFC0.0380`
‘x’ denotes don’t care					

4.6 General Exception Processing

With the exception of Reset, Soft Reset, NMI, cache error, and EJTAG Debug exceptions, which have their own special processing as described below, exceptions have the same basic processing flow:

- If the EXL bit in the *Status* register is zero, the *EPC* register is loaded with the PC at which execution will be restarted and the BD bit is set appropriately in the *Cause* register (see Table 5-22 on page 116). The value loaded into the *EPC* register is dependent on whether the processor implements the MIPS16 ASE, and whether the instruction is

in the delay slot of a branch or jump which has delay slots. Table 4-8 shows the value stored in each of the CP0 PC registers, including *EPC*. For implementations of Release 2 of the Architecture if $\text{Status}_{\text{BEV}}$ = 0, the CSS field in the *SRSCtl* register is copied to the PSS field, and the CSS value is loaded from the appropriate source.

If the EXL bit in the *Status* register is set, the *EPC* register is not loaded and the BD bit is not changed in the *Cause* register. For implementations of Release 2 of the Architecture, the *SRSCtl* register is not changed.

Table 4-8 Value Stored in EPC, ErrorEPC, or DEPC on an Exception

MIPS16 Implemented?	In Branch/Jump Delay Slot?	Value stored in EPC/ErrorEPC/DEPC
No	No	Address of the instruction
No	Yes	Address of the branch or jump instruction (PC-4)
Yes	No	Upper 31 bits of the address of the instruction, combined with the *ISA Mode* bit
Yes	Yes	Upper 31 bits of the branch or jump instruction (PC-2 in the MIPS16 ISA Mode and PC-4 in the 32-bit ISA Mode), combined with the *ISA Mode* bit

- The CE, and ExcCode fields of the *Cause* registers are loaded with the values appropriate to the exception. The CE field is loaded, but not defined, for any exception type other than a coprocessor unusable exception.
- The EXL bit is set in the *Status* register.
- The processor is started at the exception vector.

The value loaded into EPC represents the restart address for the exception and need not be modified by exception handler software in the normal case. Software need not look at the BD bit in the Cause register unless it wishes to identify the address of the instruction that actually caused the exception.

Note that individual exception types may load additional information into other registers. This is noted in the description of each exception type below.

Operation:

```
/* If Status_EXL is 1, all exceptions go through the general exception vector */
/* and neither EPC nor Cause_BD nor SRSCtl are modified */
if Status_EXL = 1 then
    vectorOffset ← 16#180
else
    if InstructionInBranchDelaySlot then
        EPC ← restartPC/* PC of branch/jump */
        Cause_BD ← 1
    else
        EPC ← restartPC                /* PC of instruction */
        Cause_BD ← 0
    endif

    /* Compute vector offsets as a function of the type of exception */
    NewShadowSet ← SRSCtl_ESS          /* Assume exception, Release 2 only */
    if ExceptionType = TLBRefill then
        vectorOffset ← 16#000
    elseif (ExceptionType = Interrupt) then
        if (Cause_IV = 0) then
            vectorOffset ← 16#180
        else
            if (Status_BEV = 1) or (IntCtl_VS = 0) then
```

```
                vectorOffset ← 16#200
            else
                if Config3_VEIC = 1 then
                    VecNum ← Cause_RIPL
                    NewShadowSet ← SRSCtl_EICSS
                else
                    VecNum ← VIntPriorityEncoder()
                    NewShadowSet ← SRSMap_IPL×4+3..IPL×4
                endif
                vectorOffset ← 16#200 + (VecNum × (IntCtl_VS || 2#00000))
            endif /* if (Status_BEV = 1) or (IntCtl_VS = 0) then */
        endif /* if (Cause_IV = 0) then */
    endif /* elseif (ExceptionType = Interrupt) then */

    /* Update the shadow set information for an implementation of */
    /* Release 2 of the architecture */
    if ((ArchitectureRevision ≥ 2) and (SRSCtl_HSS > 0) and (Status_BEV = 0) and
        (Status_ERL = 0)) then
        SRSCtl_PSS ← SRSCtl_CSS
        SRSCtl_CSS ← NewShadowSet
    endif
endif /* if Status_EXL = 1 then */

Cause_CE ← FaultingCoprocessorNumber
Cause_ExcCode ← ExceptionType
Status_EXL ← 1


/* Calculate the vector base address */
if Status_BEV = 1 then
    vectorBase ← 16#BFC0.0200
else
    if ArchitectureRevision ≥ 2 then
        /* The fixed value of EBase_31..30 forces the base to be in kseg0 or kseg1 */
        vectorBase ← EBase_31..12 || 16#000
    else
        vectorBase ← 16#8000.0000
    endif
endif

/* Exception PC is the sum of vectorBase and vectorOffset */
PC ← vectorBase_31..30 || (vectorBase_29..0 + vectorOffset_29..0)
                                /* No carry between bits 29 and 30 */
```

4.7 Debug Exception Processing

All debug exceptions have the same basic processing flow:

- The *DEPC* register is loaded with the program counter (PC) value at which execution will be restarted and the DBD bit is set appropriately in the *Debug* register. The value loaded into the *DEPC* register is the current PC if the instruction is not in the delay slot of a branch, or the PC-4 of the branch if the instruction is in the delay slot of a branch.
- The DSS, DBp, DDBL, DDBS, DIB and DINT bits (D* bits at [5:0]) in the *Debug* register are updated appropriately depending on the debug exception type.
- Halt and Doze bits in the *Debug* register are updated appropriately.
- DM bit in the *Debug* register is set to 1.

- The processor is started at the debug exception vector.

The value loaded into *DEPC* represents the restart address for the debug exception and need not be modified by the debug exception handler software in the usual case. Debug software need not look at the DBD bit in the *Debug* register unless it wishes to identify the address of the instruction that actually caused the debug exception.

A unique debug exception is indicated through the DSS, DBp, DDBL, DDBS, DIB and DINT bits (D* bits at [5:0]) in the *Debug* register.

No other CP0 registers or fields are changed due to the debug exception, thus no additional state is saved.

Operation:

```
if InstructionInBranchDelaySlot then
    DEPC ← PC-4
    Debug_DBD ← 1
else
    DEPC ← PC
    Debug_DBD ← 0
endif
Debug_D* bits at at [5:0] ← DebugExceptionType
Debug_Halt ← HaltStatusAtDebugException
Debug_Doze ← DozeStatusAtDebugException
Debug_DM ← 1
if EJTAGControlRegister_ProbTrap = 1 then
    PC ← 0xFF20_0200
else
    PC ← 0xBFC0_0480
endif
```

The same debug exception vector location is used for all debug exceptions. The location is determined by the ProbTrap bit in the EJTAG Control register (ECR), as shown in Table 4-9.

Table 4-9 Debug Exception Vector Addresses

ProbTrap bit in ECR Register	Debug Exception Vector Address
0	0xBFC0_0480
1	0xFF20_0200 in dmseg

4.8 Exceptions

The following subsections describe each of the exceptions listed in the same sequence as shown in Table 4-1.

4.8.1 Reset Exception

A reset exception occurs when the *SI_ColdReset* signal is asserted to the processor. This exception is not maskable. When a Reset exception occurs, the processor performs a full reset initialization, including aborting state machines, establishing critical state, and generally placing the processor in a state in which it can execute instructions from uncached, unmapped address space. On a Reset exception, the state of the processor is not defined, with the following exceptions:

- The *Random* register is initialized to the number of TLB entries - 1.
- The *Wired* register is initialized to zero.

- The *Config* register is initialized with its boot state.
- The RP, BEV, TS, SR, NMI, and ERL fields of the *Status* register are initialized to a specified state.
- The I, R, and W fields of the *WatchLo* register are initialized to 0.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC. Note that this value may or may not be predictable.
- PC is loaded with 0xBFC0_0000.

***Cause* Register ExcCode Value:**

None

Additional State Saved:

None

Entry Vector Used:

Reset (0xBFC0_0000)

Operation:

```
Random ← TLBEntries - 1
Wired ← 0
Config ← ConfigurationState
Status_RP ← 0
Status_BEV ← 1
Status_TS ← 0
Status_SR ← 0
Status_NMI ← 0
Status_ERL ← 1
WatchLo_I ← 0
WatchLo_R ← 0
WatchLo_W ← 0
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else
    ErrorEPC ← PC
endif
PC ← 0xBFC0_0000
```

4.8.2 Soft Reset Exception

A soft reset exception occurs when the *SI_Reset* signal is asserted to the processor. This exception is not maskable. When a soft reset exception occurs, the processor performs a subset of the full reset initialization. Although a soft reset exception does not unnecessarily change the state of the processor, it may be forced to do so in order to place the processor in a state in which it can execute instructions from uncached, unmapped address space. Since bus, cache, or other operations may be interrupted, portions of the cache, memory, or other processor state may be inconsistent. In addition to any hardware initialization required, the following state is established on a soft reset exception:

- The BEV, TS, SR, NMI, and ERL fields of the *Status* register are initialized to a specified state.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC. Note that this value may or may not be predictable.
- PC is loaded with 0xBFC0_0000.

***Cause* Register ExcCode Value:**

None

Additional State Saved:

None

Entry Vector Used:

Reset (0xBFC0_0000)

Operation:

```
Status_BEV ← 1
Status_TS ← 0
Status_SR ← 1
Status_NMI ← 0
Status_ERL ← 1
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else
    ErrorEPC ← PC
endif
PC ← 0xBFC0_0000
```

4.8.3 Debug Single Step Exception

A debug single step exception occurs after the CPU has executed one/two instructions in non-debug mode, when returning to non-debug mode after debug mode. One instruction is allowed to execute when returning to a non jump/branch instruction, otherwise two instructions are allowed to execute since the jump/branch and the instruction in the delay slot are executed as one step. Debug single step exceptions are enabled by the SSt bit in the Debug register, and are always disabled for the first one/two instructions after a DERET.

The DEPC register points to the instruction on which the debug single step exception occurred, which is also the next instruction to single step or execute when returning from debug mode. So the DEPC will not point to the instruction which has just been single stepped, but rather the following instruction. The DBD bit in the Debug register is never set for a debug single step exception, since the jump/branch and the instruction in the delay slot is executed in one step.

Exceptions occurring on the instruction(s) executed with debug single step exception enabled are taken even though debug single step was enabled. For a normal exception (other than reset), a debug single step exception is then taken on the first instruction in the normal exception handler. Debug exceptions are unaffected by single step mode, e.g. returning to a SDBBP instruction with debug single step exceptions enabled causes a debug software breakpoint exception, and the DEPC will point to the SDBBP instruction. However, returning to an instruction (not jump/branch) just before the SDBBP instruction, causes a debug single step exception with the DEPC pointing to the SDBBP instruction.

To ensure proper functionality of single step, the debug single step exception has priority over all other exceptions, except reset and soft reset.

Debug Register Debug Status Bit Set

DSS

Additional State Saved

None

Entry Vector Used

Debug exception vector

4.8.4 Debug Interrupt Exception

A debug interrupt exception is either caused by the EjtagBrk bit in the *EJTAG Control register* (controlled through the TAP), or caused by the debug interrupt request signal to the CPU.

The debug interrupt exception is an asynchronous debug exception which is taken as soon as possible, but with no specific relation to the executed instructions. The *DEPC* register is set to the instruction where execution should continue after the debug handler is through. The DBD bit is set based on whether the interrupted instruction was executing in the delay slot of a branch.

Debug Register Debug Status Bit Set

DINT

Additional State Saved

None

Entry Vector Used

Debug exception vector

4.8.5 Non-Maskable Interrupt (NMI) Exception

A non maskable interrupt exception occurs when the *SI_NMI* signal is asserted to the processor. *SI_NMI* is an edge sensitive signal - only one NMI exception will be taken each time it is asserted. An NMI exception occurs only at instruction boundaries, so it does not cause any reset or other hardware initialization. The state of the cache, memory, and other processor states are consistent and all registers are preserved, with the following exceptions:

- The BEV, TS, SR, NMI, and ERL fields of the *Status* register are initialized to a specified state.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC.
- PC is loaded with 0xBFC0_0000.

***Cause* Register ExcCode Value:**

None

Additional State Saved:

None

Entry Vector Used:

Reset (0xBFC0_0000)

Operation:

$\text{Status}_{\text{BEV}} \leftarrow 1$
$\text{Status}_{\text{TS}} \leftarrow 0$
$\text{Status}_{\text{SR}} \leftarrow 0$
$\text{Status}_{\text{NMI}} \leftarrow 1$
$\text{Status}_{\text{ERL}} \leftarrow 1$

```
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else
    ErrorEPC ← PC
endif
PC ← 0xBFC0_0000
```

4.8.6 Machine Check Exception (4KEc core)

A machine check exception occurs when the processor detects an internal inconsistency. The following condition causes a machine check exception:

- The detection of multiple matching entries in the TLB (4KEc core only). The core detects this condition on a TLB write and prevents the write from being completed. The TS bit in the *Status* register is set to indicate this condition. This bit is only a status flag and does not affect the operation of the device. Software clears this bit at the appropriate time. This condition is resolved by flushing the conflicting TLB entries. The TLB write can then be completed.

***Cause* Register ExcCode Value:**

MCheck

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.7 Interrupt Exception

The interrupt exception occurs when one or more of the six hardware, two software, or timer interrupt requests is enabled by the *Status* register and the interrupt input is asserted. See Section 4.3, "Interrupts" on page 55 for more details about the processing of interrupts.

XXX Is this paragraph still relevant? XXX The delay from assertion of an unmasked interrupt to the fetch of the first instructions at the exception vector is a minimum of 5 clock cycles. More may be needed if a committed instruction has to complete, before the exception can be taken; i.e., an uncached load that has started on the bus must wait complete before the interrupt exception can be taken.

Register ExcCode Value:

Int

Additional State Saved:

Table 4-10 Register States an Interrupt Exception

Register State	Value
$Cause_{IP}$	indicates the interrupts that are pending.

Entry Vector Used:

See Section 4.3.2, "Generation of Exception Vector Offsets for Vectored Interrupts" on page 63 for the entry vector used, depending on the interrupt mode the processor is operating in.

4.8.8 Debug Instruction Break Exception

A debug instruction break exception occurs when an instruction hardware breakpoint matches an executed instruction. The *DEPC* register and DBD bit in the *Debug* register indicate the instruction that caused the instruction hardware breakpoint to match. This exception can only occur if instruction hardware breakpoints are implemented.

Debug Register Debug Status Bit Set:

DIB

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

4.8.9 Watch Exception — Instruction Fetch or Data Access

The Watch facility provides a software debugging vehicle by initiating a watch exception when an instruction or data reference matches the address information stored in the *WatchHi* and *WatchLo* registers. A Watch exception is taken immediately if the EXL and ERL bits of the *Status* register are both zero and the DM bit of the *Debug* is also zero. If any of those bits is a one at the time that a watch exception would normally be taken, then the WP bit in the *Cause* register is set, and the exception is deferred until all three bits are zero. Software may use the WP bit in the *Cause* register to determine if the *EPC* register points at the instruction that caused the watch exception, or if the exception actually occurred while in kernel mode.

The Watch exception can occur on either an instruction fetch or a data access. Watch exceptions that occur on an instruction fetch have a higher priority than watch exceptions that occur on a data access.

Register ExcCode Value:

WATCH

Additional State Saved:

Table 4-11 Register States on a Watch Exception

Register State	Value
$Cause_{WP}$	Indicates that the watch exception was deferred until after $Status_{EXL}$, $Status_{ERL}$, and $Debug_{DM}$ were zero. This bit directly causes a watch exception, so software must clear this bit as part of the exception handler to prevent a watch exception loop at the end of the current handler execution.

Entry Vector Used:

General exception vector (offset 0x180)

4.8.10 Address Error Exception — Instruction Fetch/Data Access

An address error exception occurs on an instruction or data access when an attempt is made to execute one of the following:

- Fetch an instruction, load a word, or store a word that is not aligned on a word boundary
- Load or store a halfword that is not aligned on a halfword boundary
- Reference the kernel address space from user mode

Note that in the case of an instruction fetch that is not aligned on a word boundary, PC is updated before the condition is detected. Therefore, both EPC and BadVAddr point to the unaligned instruction address. In the case of a data access the exception is taken if either an unaligned address or an address that was inaccessible in the current processor mode was referenced by a load or store instruction.

***Cause* Register ExcCode Value:**

ADEL: Reference was a load or an instruction fetch

ADES: Reference was a store

Additional State Saved:

Table 4-12 CP0 Register States on an Address Exception Error

Register State	Value
BadVAddr	failing address
Context_{VPN2}	UNPREDICTABLE
EntryHi_{VPN2}	UNPREDICTABLE
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

4.8.11 TLB Refill Exception — Instruction Fetch or Data Access (4KEc core only)

During an instruction fetch or data access, a TLB refill exception occurs when no TLB entry matches a reference to a mapped address space and the EXL bit is 0 in the *Status* register. Note that this is distinct from the case in which an entry matches but has the valid bit off. In that case, a TLB Invalid exception occurs.

***Cause* Register ExcCode Value:**

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

Additional State Saved:

Table 4-13 CP0 Register States on a TLB Refill Exception

Register State	Value
BadVAddr	failing address.
Context	The BadVPN2 field contains $VA_{31:13}$ of the failing address.
EntryHi	The VPN2 field contains $VA_{31:13}$ of the failing address; the ASID field contains the ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

TLB refill vector (offset 0x000) if $\text{Status}_{EXL} = 0$ at the time of exception;

general exception vector (offset 0x180) if $\text{Status}_{EXL} = 1$ at the time of exception

4.8.12 TLB Invalid Exception — Instruction Fetch or Data Access (4KEc core only)

During an instruction fetch or data access, a TLB invalid exception occurs in one of the following cases:

- No TLB entry matches a reference to a mapped address space; and the EXL bit is 1 in the *Status* register.
- A TLB entry matches a reference to a mapped address space, but the matched entry has the valid bit off.
- The virtual address is greater than or equal to the bounds address in a FM-based MMU.

***Cause* Register ExcCode Value:**

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

Additional State Saved:

Table 4-14 CP0 Register States on a TLB Invalid Exception

Register State	Value
BadVAddr	failing address
Context	The BadVPN2 field contains $VA_{31:13}$ of the failing address.
EntryHi	The VPN2 field contains $VA_{31:13}$ of the failing address; the ASID field contains the ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

4.8.13 Bus Error Exception — Instruction Fetch or Data Access

A bus error exception occurs when an instruction or data access makes a bus request (due to a cache miss or an uncacheable reference) and that request terminates in an error. The bus error exception can occur on either an instruction fetch or a data access. Bus error exceptions that occur on an instruction fetch have a higher priority than bus error exceptions that occur on a data access.

Bus errors taken on the requested (critical) word of an instruction fetch or data load are precise. Other bus errors, such as stores or non-critical words of a burst read, can be imprecise. These errors are taken when the *EB_RBErr* or *EB_WBErr* signals are asserted and may occur on an instruction that was not the source of the offending bus cycle.

***Cause* Register ExcCode Value:**

IBE: Error on an instruction reference

DBE: Error on a data reference

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.14 Debug Software Breakpoint Exception

A debug software breakpoint exception occurs when an SDBBP instruction is executed. The *DEPC* register and DBD bit in the *Debug* register will indicate the SDBBP instruction that caused the debug exception.

Debug Register Debug Status Bit Set:

DBp

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

4.8.15 Execution Exception — System Call

The system call exception is one of the nine execution exceptions. All of these exceptions have the same priority. A system call exception occurs when a SYSCALL instruction is executed.

***Cause* Register ExcCode Value:**

Sys

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.16 Execution Exception — Breakpoint

The breakpoint exception is one of the nine execution exceptions. All of these exceptions have the same priority. A breakpoint exception occurs when a BREAK instruction is executed.

***Cause* Register ExcCode Value:**

Bp

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.17 Execution Exception — Reserved Instruction

The reserved instruction exception is one of the nine execution exceptions. All of these exceptions have the same priority. A reserved instruction exception occurs when a reserved or undefined major opcode or function field is executed. This includes Coprocessor 2 instructions which are decoded reserved in the Coprocessor 2.

***Cause* Register ExcCode Value:**

RI

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.18 Execution Exception — Coprocessor Unusable

The coprocessor unusable exception is one of the nine execution exceptions. All of these exceptions have the same priority. A coprocessor unusable exception occurs when an attempt is made to execute a coprocessor instruction for one of the following:

- a corresponding coprocessor unit that has not been marked usable by setting its CU bit in the *Status* register
- CP0 instructions, when the unit has not been marked usable, and the processor is executing in user mode

***Cause* Register ExcCode Value:**

CpU

Additional State Saved:

Table 4-15 Register States on a Coprocessor Unusable Exception

Register State	Value
$Cause_{CE}$	unit number of the coprocessor being referenced

Entry Vector Used:

General exception vector (offset 0x180)

4.8.19 Execution Exception — Coprocessor 2 Exception

The Coprocessor 2 exception is one of the nine execution exceptions. All of these exceptions have the same priority. A Coprocessor 2 exception occurs when a valid Coprocessor 2 instruction cause a general exception in the Coprocessor 2.

***Cause* Register ExcCode Value:**

C2E

Additional State Saved:

Depending on the Coprocessor 2 implementation, additional state information of the exception can be saved in a Coprocessor 2 control register.

Entry Vector Used:

General exception vector (offset 0x180)

4.8.20 Execution Exception — Implementation-Specific 1 exception

The Implementation-Specific 1 exception is one of the nine execution exceptions. All of these exceptions have the same priority. An implementation-specific 1 exception occurs when a valid coprocessor 2 instruction cause an implementation-specific 1 exception in the Coprocessor 2.

Cause **Register ExcCode Value:**

IS1

Additional State Saved:

Depending on the coprocessor 2 implementation, additional state information of the exception can be saved in a coprocessor 2 control register.

Entry Vector Used:

General exception vector (offset 0x180)

4.8.21 Execution Exception — Implementation Specific 2 exception

The Implementation-Specific 2 exception is one of the nine execution exceptions. All of these exceptions have the same priority. An implementation-specific 2 exception occurs when a valid Coprocessor 2 instruction cause an implementation-specific 2 exception in the Coprocessor 2.

Cause **Register ExcCode Value:**

IS2

Additional State Saved:

Depending on the Coprocessor 2 implementation, additional state information of the exception can be saved in a Coprocessor 2 control register.

Entry Vector Used:

General exception vector (offset 0x180)

4.8.22 Execution Exception — Integer Overflow

The integer overflow exception is one of the nine execution exceptions. All of these exceptions have the same priority. An integer overflow exception occurs when selected integer instructions result in a 2's complement overflow.

Cause **Register ExcCode Value:**

Ov

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.23 Execution Exception — Trap

The trap exception is one of the nine execution exceptions. All of these exceptions have the same priority. A trap exception occurs when a trap instruction results in a TRUE value.

Cause **Register ExcCode Value:**

Tr

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

4.8.24 Debug Data Break Exception

A debug data break exception occurs when a data hardware breakpoint matches the load/store transaction of an executed load/store instruction. The *DEPC* register and DBD bit in the *Debug* register will indicate the load/store instruction that caused the data hardware breakpoint to match. The load/store instruction that caused the debug exception has not completed e.g. not updated the register file, and the instruction can be re-executed after returning from the debug handler.

Debug Register Debug Status Bit Set:

DDBL for a load instruction or DDBS for a store instruction

Additional State Saved:

None

Entry Vector Used:

Debug exception vector

4.8.25 TLB Modified Exception — Data Access (4KEc core only)

During a data access, a TLB modified exception occurs on a store reference to a mapped address if the following condition is true:

- The matching TLB entry is valid, but not dirty.

Cause **Register ExcCode Value:**

Mod

Additional State Saved:

Table 4-16 Register States on a TLB Modified Exception

Register State	Value
BadVAddr	failing address
Context	The BadVPN2 field contains $VA_{31:13}$ of the failing address.
EntryHi	The VPN2 field contains $VA_{31:13}$ of the failing address; the ASID field contains the ASID of the reference that missed.
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

4.9 Exception Handling and Servicing Flowcharts

The remainder of this chapter contains flowcharts for the following exceptions and guidelines for their handlers:

- General exceptions and their exception handler
- TLB miss exception and their exception handler
- Reset, soft reset and NMI exceptions, and a guideline to their handler.
- Debug exceptions

Generally speaking, the exceptions are handled by hardware; the exceptions are then serviced by software. Note that unexpected debug exceptions to the debug exception vector at 0xBFC0_0200 may be viewed as a reserved instruction since uncontrolled execution of an SDBBP instruction caused the exception. The DERET instruction must be used at return from the debug exception handler, in order to leave debug mode and return to non-debug mode. The DERET instruction returns to the address in the *DEPC* register.

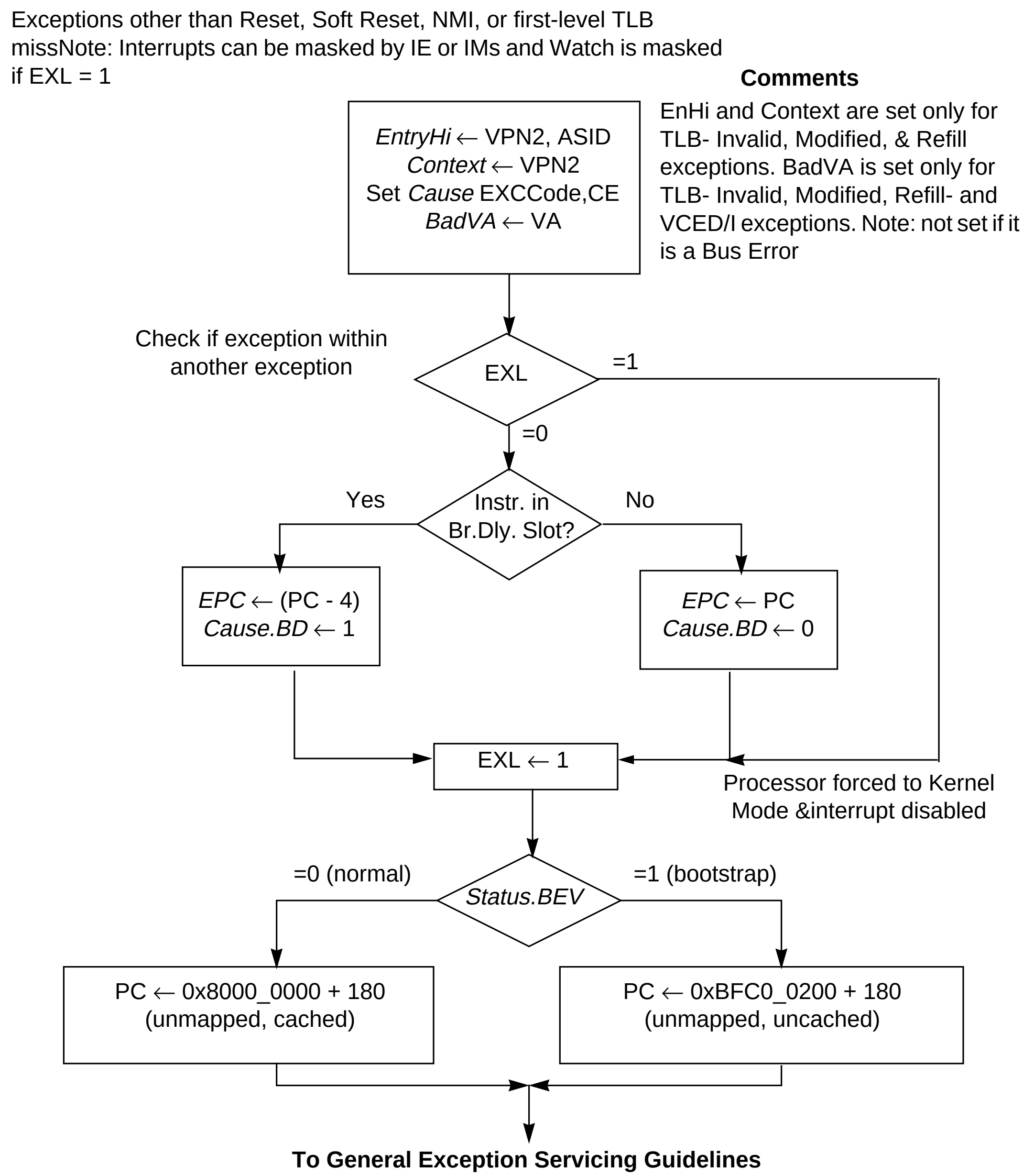


Figure 4-3 General Exception Handler (HW)

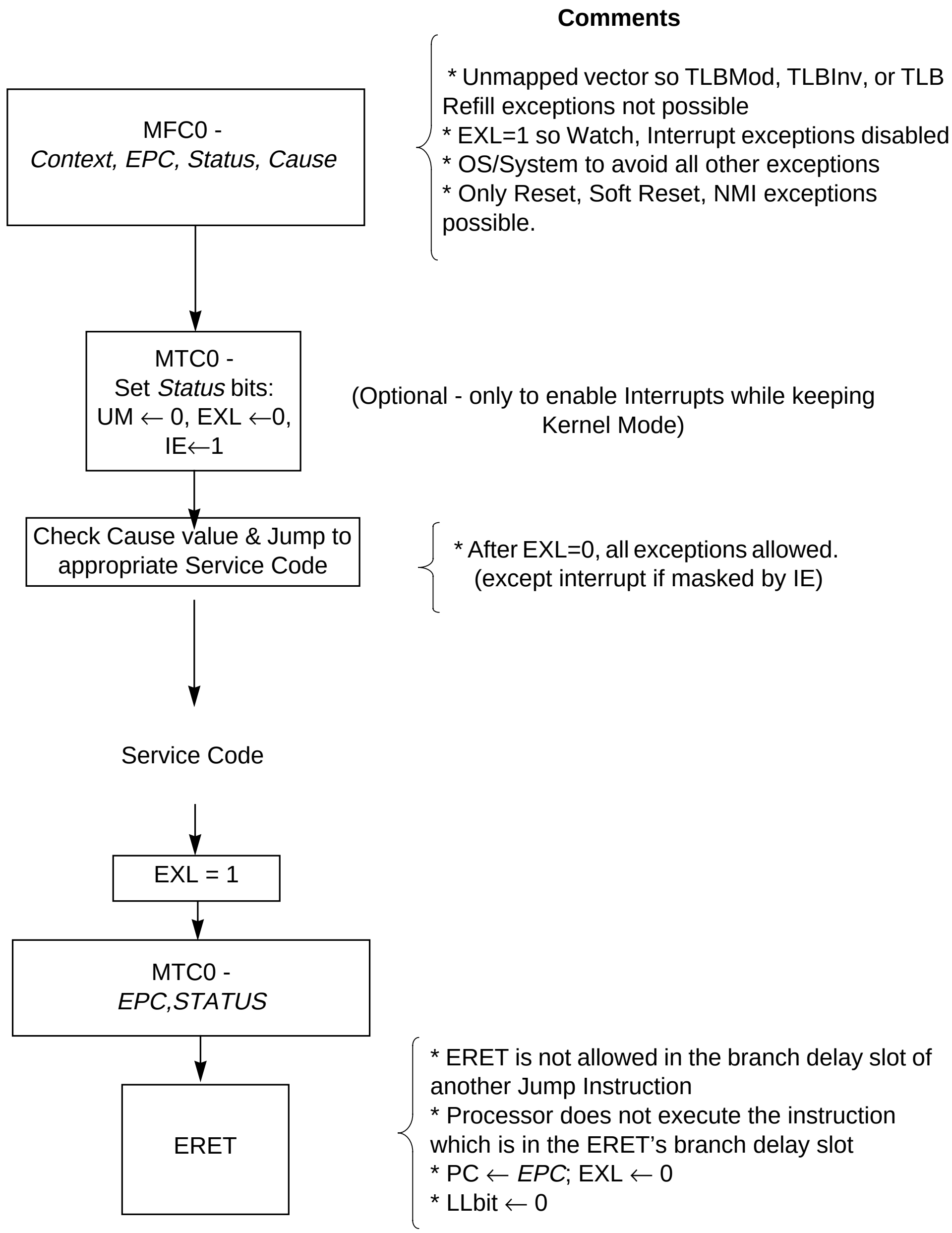


Figure 4-4 General Exception Servicing Guidelines (SW)

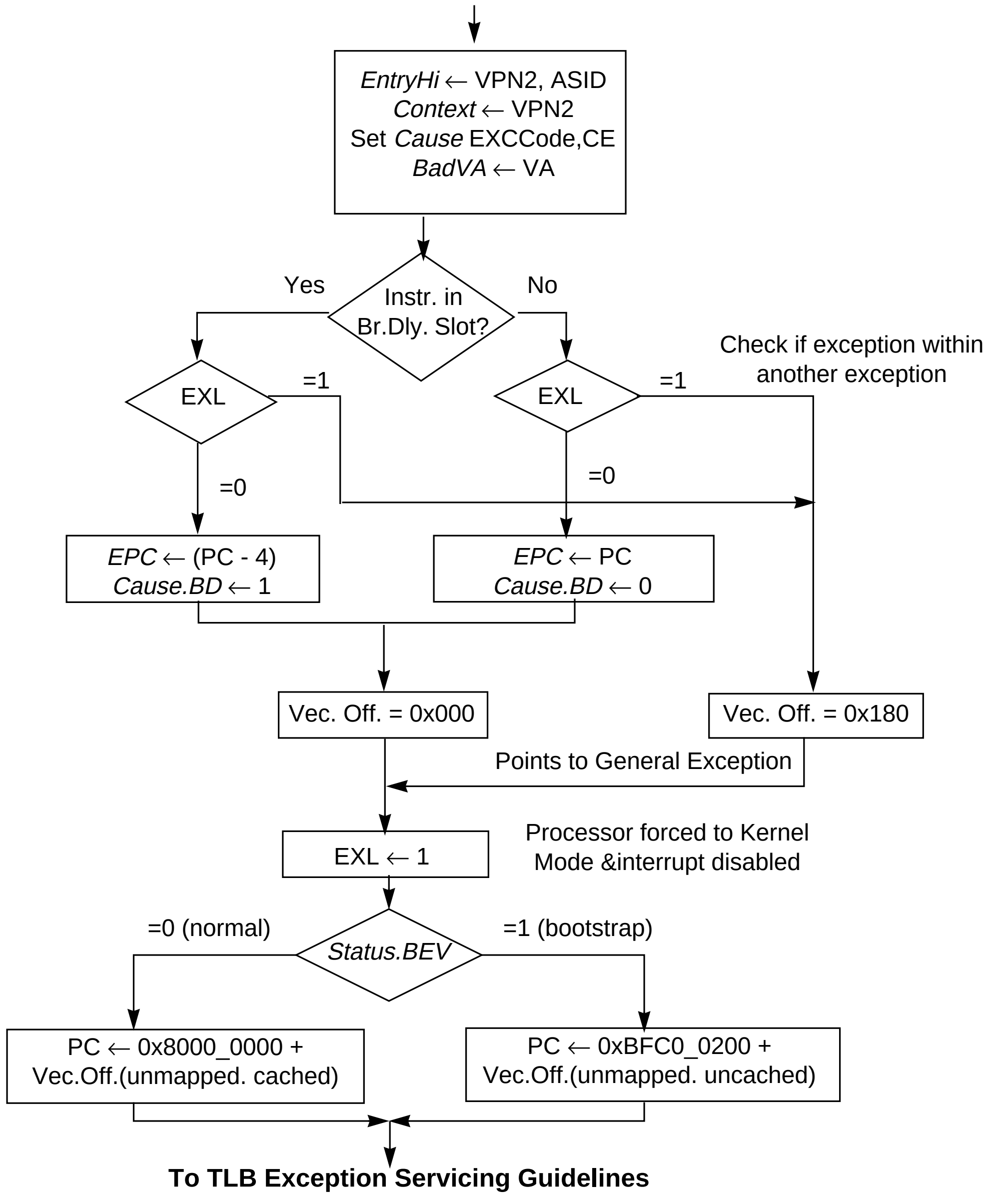


Figure 4-5 TLB Miss Exception Handler (HW) — 4KEc Core

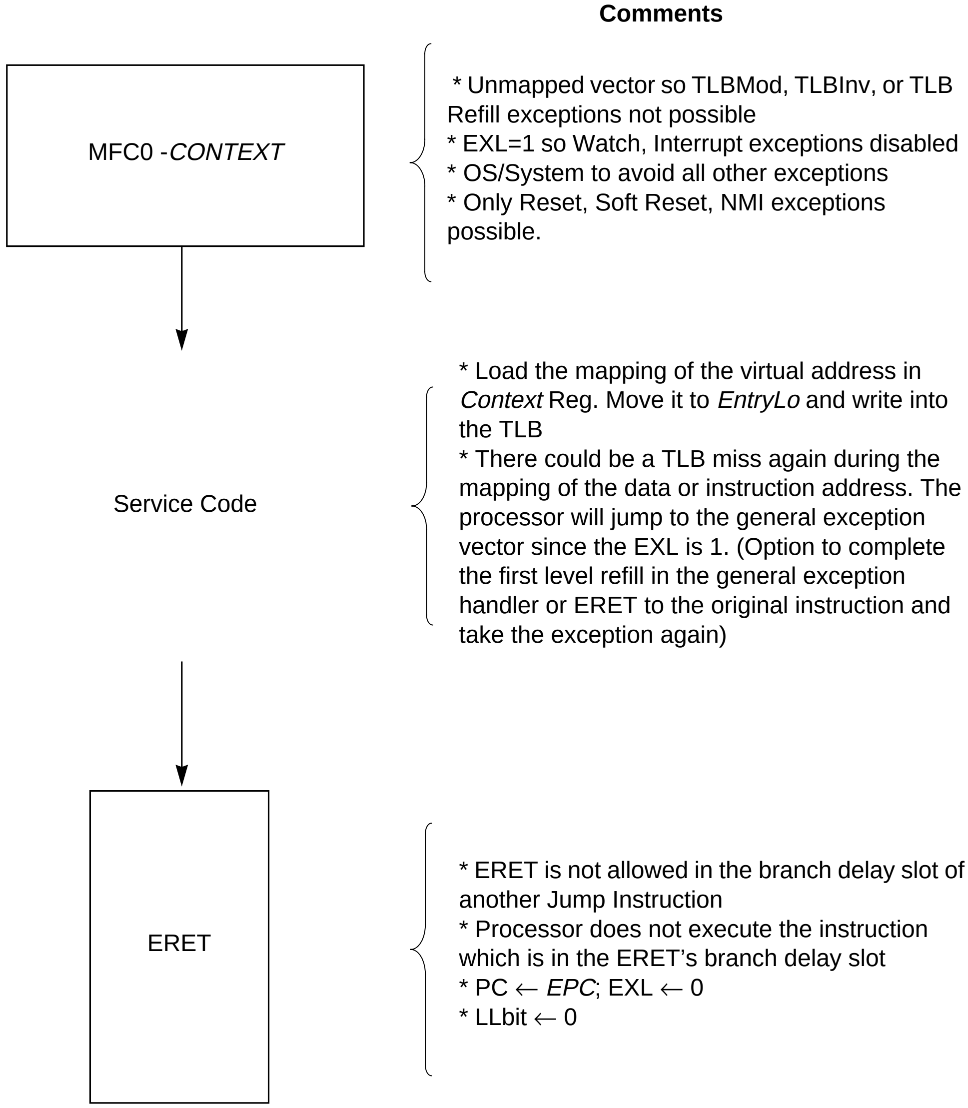


Figure 4-6 TLB Exception Servicing Guidelines (SW) — 4KEc Core

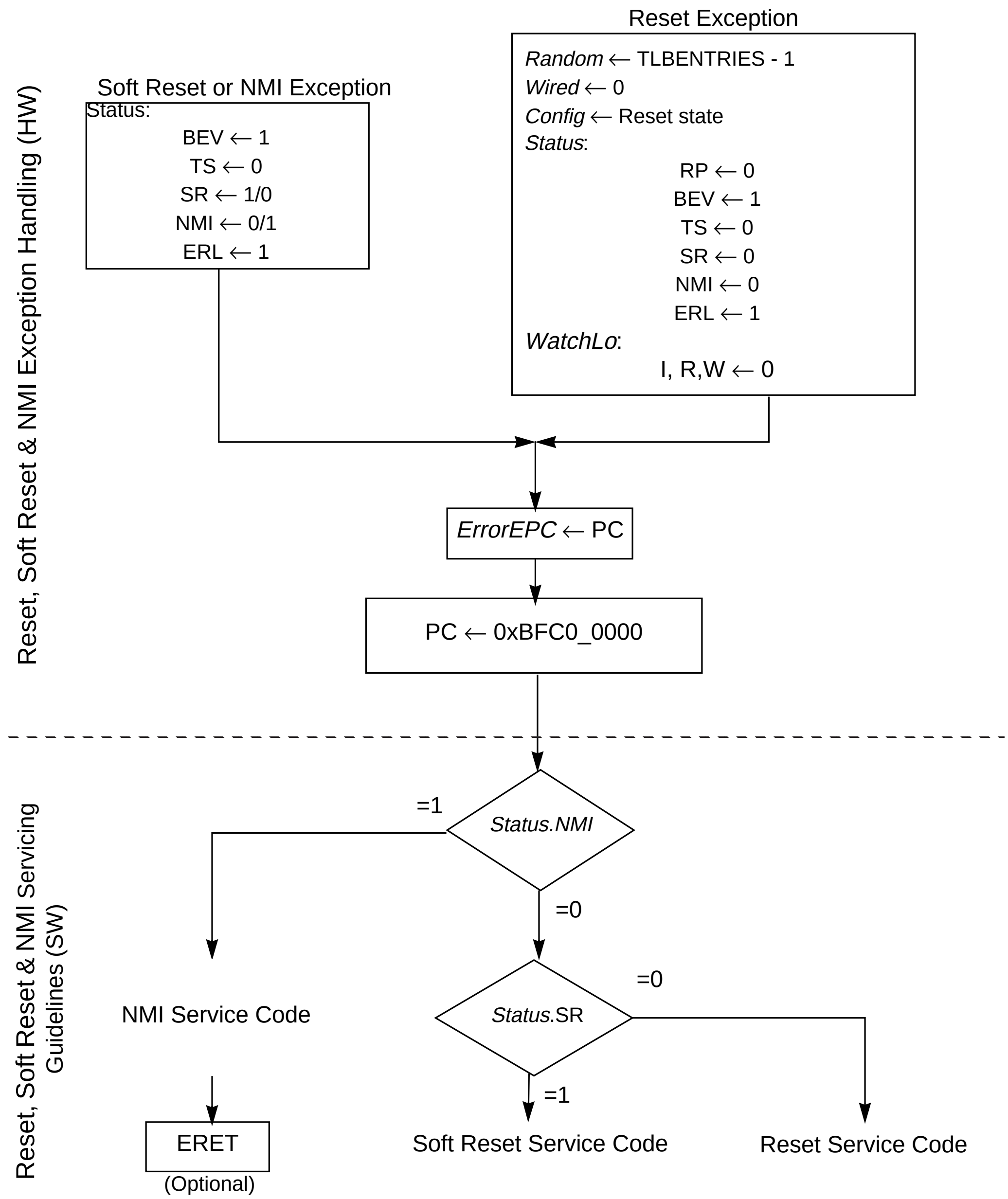


Figure 4-7 Reset, Soft Reset and NMI Exception Handling and Servicing Guidelines

Chapter 5

CP0 Registers

The System Control Coprocessor (CP0) provides the register interface to the MIPS32™ 4KE™ processor core and supports memory management, address translation, exception handling, and other privileged operations. Each CP0 register has a unique number that identifies it; this number is referred to as the *register number*. For instance, the *PageMask* register is register number 5. For more information on the EJTAG registers, refer to Chapter 9, “EJTAG Debug Support.”.

After updating a CP0 register there is a hazard period of zero or more instructions from the update instruction (MTC0) and until the effect of the update has taken place in the core. Refer to Chapter 11, “MIPS32™ 4KE™ Processor Core Instructions,” for further details on CP0 hazards.

The current chapter contains the following sections:

5.1 CP0 Register Summary

Table 5-1 lists the CP0 registers in numerical order. The individual registers are described throughout this chapter. Where more than one registers shares the same register number at different values of the "sel" field of the instruction, their names are listed using a slash (/) as separator.

Table 5-1 CP0 Registers

Register Number	Register Name	Function
0	Index[3]	Index into the TLB array (4KEc core). This register is reserved in the 4KEp and 4KEm cores.
1	Random[3]	Randomly generated index into the TLB array (4KEc core). This register is reserved in the 4KEp and 4KEm cores.
2	EntryLo0[3]	Low-order portion of the TLB entry for even-numbered virtual pages (4KEc core). This register is reserved in the 4KEp and 4KEm cores.
3	EntryLo1[3]	Low-order portion of the TLB entry for odd-numbered virtual pages (4KEc core). This register is reserved in the 4KEp and 4KEm cores.
4	Context[1]	Pointer to page table entry in memory (4KEc core). This register is reserved in the 4KEp and 4KEm cores.
5	PageMask/ PageGrain[3]	PageMask controls the variable page sizes in TLB entries. PageGrain enables support of 1KB pages in the TLB. These registers are defined for the 4KEc core only, and reserved in the 4KEp and 4KEm cores.
6	Wired[3]	Controls the number of fixed ("wired") TLB entries (4KEc core). This register is reserved in the 4KEp and 4KEm cores.
7	HWREna	Enables access via the RDHWR instruction to selected hardware registers in non-privileged mode.
8	BadVAddr[1]	Reports the address for the most recent address-related exception.
9	Count[1]	Processor cycle count.
10	EntryHi[3]	High-order portion of the TLB entry (4KEc core). This register is reserved in the 4KEp and 4KEm cores.
11	Compare[1]	Timer interrupt control.
12	Status/ IntCtl/ SRSCtl/ SRSMap[1]	Processor status and control; interrupt control; and shadow set control.
13	Cause[1]	Cause of last exception.
14	EPC[1]	Program counter at last exception.
15	PRId/ EBase	Processor identification and revision; exception base address.

Table 5-1 CP0 Registers (Continued)

Register Number	Register Name	Function
16	Config/ Config1/ Config2/ Config3	Configuration registers.
17	LLAddr	Load linked address.
18	WatchLo[1]	Low-order watchpoint address.
19	WatchHi[1]	High-order watchpoint address.
20 - 22	Reserved	Reserved
23	Debug/ TraceControl/ TraceControl2/ UserTraceData/ TraceBPC[2]	Debug control/exception status and EJTAG trace control.
24	DEPC[2]	Program counter at last debug exception.
25	Reserved	Reserved
26	ErrCtl	Software test enable of way-select and Data RAM arrays for I-Cache and D-Cache.
27	Reserved	Reserved
28	TagLo/DataLo	Low-order portion of cache tag interface.
29	Reserved	Reserved
30	ErrorEPC[1]	Program counter at last error.
31	DeSAVE[2]	Debug handler scratchpad register.
Note: 1. Registers used in exception processing. Note: 2. Registers used in debug. Note: 3. Registers used in memory management.		

5.2 CP0 Register Descriptions

The CP0 registers provide the interface between the ISA and the architecture. Each register is discussed below, with the registers presented in numerical order, first by register number, then by select field number.

For each register described below, field descriptions include the read/write properties of the field, and the reset state of the field. For the read/write properties of the field, the following notation is used:

Table 5-2 CP0 Register Field Types

<table>
<tr><th>Read/Write Notation</th><th>Hardware Interpretation</th><th>Software Interpretation</th></tr>
<tr><td>R/W</td><td colspan="2">A field in which all bits are readable and writable by software and, potentially, by hardware.

Hardware updates of this field are visible by software reads. Software updates of this field are visible by hardware reads.

If the reset state of this field is “Undefined,” either software or hardware must initialize the value before the first read will return a predictable value. This should not be confused with the formal definition of UNDEFINED behavior.</td></tr>
<tr><td>R</td><td>A field that is either static or is updated only by hardware.

If the Reset State of this field is either “0” or “Preset”, hardware initializes this field to zero or to the appropriate state, respectively, on powerup.

If the Reset State of this field is “Undefined”, hardware updates this field only under those conditions specified in the description of the field.</td><td>A field to which the value written by software is ignored by hardware. Software may write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware.

If the Reset State of this field is “Undefined,” software reads of this field result in an UNPREDICTABLE value except after a hardware update done under the conditions specified in the description of the field.</td></tr>
<tr><td>W</td><td colspan="2">A field that can be written by software but which can not be read by software.

Software reads of this field will return an UNDEFINED value.</td></tr>
<tr><td>0</td><td>A field that hardware does not update, and for which hardware can assume a zero value.</td><td>A field to which the value written by software must be zero. Software writes of non-zero values to this field may result in UNDEFINED behavior of the hardware. Software reads of this field return zero as long as all previous software writes are zero.

If the Reset State of this field is “Undefined,” software must write this field with zero before it is guaranteed to read as zero.</td></tr>
</table>

5.2.1 *Index* Register (CP0 Register 0, Select 0)

The *Index* register is a 32-bit read/write register that contains the index used to access the TLB for TLBP, TLBR, and TLBWI instructions. The width of the index field is implementation-dependent as a function of the number of TLB entries that are implemented. The minimum value for TLB-based MMUs is *Ceiling*(Log_2*(TLBEntries))*.

The operation of the processor is UNDEFINED if a value greater than or equal to the number of TLB entries is written to the *Index* register.

This register is only valid with the TLB (4KEc core). It is reserved if the FM is implemented (4KEm and 4KEp).

Figure 5-1 *Index* Register Format

31	30 4	3 0
P	0	Index

Table 5-3 Index Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
P	31	Probe Failure. Set to 1 when the previous TLBProbe (TLBP) instruction failed to find a match in the TLB.	R	Undefined
0	30:4	Must be written as zeros; returns zeros on reads.	0	0
Index	3:0	Index to the TLB entry affected by the TLBRead and TLBWrite instructions.	R/W	Undefined

5.2.2 *Random* Register (CP0 Register 1, Select 0)

The *Random* register is a read-only register whose value is used to index the TLB during a TLBWR instruction. The width of the Random field is calculated in the same manner as that described for the *Index* register above.

The value of the register varies between an upper and lower bound as follow:

- A lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the *Wired* register). The entry indexed by the *Wired* register is the first entry available to be written by a TLB Write Random operation.
- An upper bound is set by the total number of TLB entries minus 1.

The *Random* register is decremented by one almost every clock, wrapping after the value in the *Wired* register is reached. To enhance the level of randomness and reduce the possibility of a live lock condition, an LFSR register is used that prevents the decrement pseudo-randomly.

The processor initializes the *Random* register to the upper bound on a Reset exception and when the *Wired* register is written.

This register is only valid with the TLB (4KEc core). It is reserved if the FM is implemented (4KEm and 4KEp).

Figure 5-2 *Random* Register Format

31 4	3 0
0	Random

Table 5-4 *Random* Register Field Descriptions

Fields				
Name	**Bit(s)**	**Description**	**Read/ Write**	**Reset State**
0	31:4	Must be written as zero; returns zero on reads.	0	0
Random	3:0	TLB Random Index	R	TLB Entries - 1

5.2.3 *EntryLo0* and *EntryLo1* Registers (CP0 Registers 2 and 3, Select 0)

The pair of *EntryLo* registers act as the interface between the TLB and the TLBR, TLBWI, and TLBWR instructions. For a TLB-based MMU, *EntryLo0* holds the entries for even pages and *EntryLo1* holds the entries for odd pages.

The contents of the *EntryLo0* and *EntryLo1* registers are undefined after an address error, TLB invalid, TLB modified, or TLB refill exception.

These registers are only valid with the TLB (4KEc core). They are reserved if the FM is implemented (4KEm and 4KEp).

Figure 5-3 *EntryLo0, EntryLo1* Register Format

31 30	29 26	25 6	5 3	2	1	0
R	0	PFN	C	D	V	G

Table 5-5 *EntryLo0, EntryLo1* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
R	31:30	Reserved. Should be ignored on writes; returns zero on reads.	R	0
0	29:26	These 4 bits are normally part of the PFN, however, since the core supports only 32 bits of physical address, the PFN is only 20 bits wide; therefore, bits 29:26 of this register must be written with zeros.	R/W	0
PFN	25:6	Page Frame Number. Contributes to the definition of the high-order bits of the physical address. If the processor is enabled to support 1KB pages ($Config3_{SP}$ = 1 and $PageGrain_{ESP}$ = 1), the PFN field corresponds to bits 29..10 of the physical address (the field is shifted left by 2 bits relative to the Release 1 definition to make room for $PA_{11..10}$). If the processor is not enabled to support 1KB pages ($Config3_{SP}$ = 0 or $PageGrain_{ESP}$ = 0), the PFN field corresponds to bits 31..12 of the physical address.	R/W	Undefined
C	5:3	Coherency attribute of the page. See Table 5-6.	R/W	Undefined
D	2	"Dirty" or write-enable bit, indicating that the page has been written, and/or is writable. If this bit is a one, then stores to the page are permitted. If this bit is a zero, then stores to the page cause a TLB Modified exception.	R/W	Undefined
V	1	Valid bit, indicating that the TLB entry, and thus the virtual page mapping are valid. If this bit is a one, then accesses to the page are permitted. If this bit is a zero, then accesses to the page cause a TLB Invalid exception.	R/W	Undefined
G	0	Global bit. On a TLB write, the logical AND of the G bits in both the EntryLo0 and EntryLo1 register becomes the G bit in the TLB entry. If the TLB entry G bit is a one, then the ASID comparisons are ignored during TLB matches. On a read from a TLB entry, the G bits of both EntryLo0 and EntryLo1 reflect the state of the TLB G bit.	R/W	Undefined

Table 5-6 lists the encoding of the C field of the *EntryLo0* and *EntryLo1* registers and the K0 field of the *Config* register.

Table 5-6 Cache Coherency Attributes

C[5:3] Value	Cache Coherency Attribute
0	Cacheable, noncoherent, write-through, no write allocate
1	Cacheable, noncoherent, write-through, write allocate
3*, 4, 5, 6	Cacheable, noncoherent, write-back, write allocate
2*, 7	Uncached
Note: * These two values are required by the MIPS32 architecture. Only values 0, 1, 2 and 3 are used in a 4KE core. For example, values 4, 5 and 6 are not used and are mapped to 3. The value 7 is not used and is mapped to 2. Note that these values do have meaning in other MIPS Technologies processor implementations. Refer to the MIPS32 specification for more information.	

5.2.4 *Context* Register (CP0 Register 4, Select 0)

The *Context* register is a read/write register containing a pointer to an entry in the page table entry (PTE) array. This array is an operating system data structure that stores virtual-to-physical translations. During a TLB miss, the operating system loads the TLB with the missing translation from the PTE array. The *Context* register duplicates some of the information provided in the *BadVAddr* register but is organized in such a way that the operating system can directly reference an 8-byte page table entry (PTE) in memory.

A TLB exception (TLB Refill, TLB Invalid, or TLB Modified) causes bits $VA_{31:13}$ of the virtual address to be written into the BadVPN2 field of the *Context* register. The PTEBase field is written and used by the operating system.

The BadVPN2 field of the *Context* register is not defined after an address error exception.

Figure 5-4 *Context* Register Format

31 23	22 4	3 2 1 0
PTEBase	BadVPN2	0

Table 5-7 *Context* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
PTEBase	31:23	This field is for use by the operating system and is normally written with a value that allows the operating system to use the Context Register as a pointer into the current PTE array in memory.	R/W	Undefined
BadVPN2	22:4	This field is written by hardware on a TLB miss. It contains bits $VA_{31:13}$ of the virtual address that missed.	R	Undefined
0	3:0	Must be written as zero; returns zero on reads.	0	0

5.2.5 *PageMask* Register (CP0 Register 5, Select 0)

The *PageMask* register is a read/write register used for reading from and writing to the TLB. It holds a comparison mask that sets the variable page size for each TLB entry, as shown in Table 5-9. Figure 5-5 shows the format of the *PageMask* register; Table 5-8 describes the *PageMask* register fields.

This register is only valid with the TLB (4KEc core). It is reserved if the FM is implemented (4KEm and 4KEp).

Figure 5-5 *PageMask* Register Format

31..29	28..13	12..11	10..0
0	Mask	MaskX	0

Table 5-8 *PageMask* Register Field Descriptions

Fields: Name	Fields: Bits	Description	Read/ Write	Reset State
Mask	28..13	The Mask field is a bit mask in which a “1” bit indicates that the corresponding bit of the virtual address should not participate in the TLB match.	R/W	Undefined
MaskX	12..11	In Release 2 of the Architecture, the MaskX field is an extension to the Mask field to support 1KB pages with definition and action analogous to that of the Mask field, defined above. If 1KB pages are enabled ($Config3_{SP}$ = 1 and $PageGrain_{ESP}$ = 1), these bits are writable and readable, and their values are copied to and from the TLB entry on a TLB write or read, respectivly. If 1KB pages are not enabled ($Config3_{SP}$ = 0 or $PageGrain_{ESP}$ = 0), these bits are not writable, return zero on read, and the effect on the TLB entry on a write is as if they were written with the value 2#11. In Release 1 of the Architecture, these bits must be written as zero, return zero on read, and have no effect on the virtual address translation.	R/W	0 (See Description)
0	31..29, 10..0	Ignored on write; returns zero on read.	R	0

Table 5-9 Values for the Mask and MaskX[1] Fields of the *PageMask* Register

Page Size	Bit 28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12[1]	11[1]
1 KByte	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4 KBytes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
16 KBytes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
64 KBytes	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
256 KBytes	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Table 5-9 Values for the Mask and MaskX[1] Fields of the *PageMask* Register

Page Size	Bit																	
	28	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12[1]**	**11[1]**
1 MByte	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
4 MByte	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
16 MByte	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
64 MByte	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
256 MByte	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

1. $PageMask_{12..11}$ = $PaskMask_{MaskX}$ exists only on implementations of Release 2 of the architecture and are treated as if they had the value 2#11 if 1K pages are not enabled ($Config3_{SP}$ = 0 or $PageGrain_{ESP}$ = 0).

It is implementation dependent how many of the encodings described in Table 5-9 are implemented. All processors must implement the 4KB page size. If a particular page size encoding is not implemented by a processor, a read of the *PageMask* register must return zeros in all bits that correspond to encodings that are not implemented, thereby potentially returning a value different than that written by software.

Software may determine which page sizes are supported by writing all ones to the *PageMask* register, then reading the value back. If a pair of bits reads back as ones, the processor implements that page size. The operation of the processor is **UNDEFINED** if software loads the Mask field with a value other than one of those listed in Table 5-9, even if the hardware returns a different value on read. Hardware may depend on this requirement in implementing hardware structures.

5.2.6 *PageGrain* Register (CP0 Register 5, Select 1)

The *PageGrain* register is a read/write register used for enabling 1KB page support. It is used for reading from and writing to the TLB.

The contents of the *PageGrain* register are not reflected in the contents of the TLB; therefore, the TLB must be flushed before any change to the PageGrain register is made. Behavior is UNDEFINED if a value other than those listed is used.

This register is only valid with the TLB (4KEc core). It is reserved if the FM is implemented (4KEm and 4KEp).

Figure 5-6 *PageGrain* Register Format

31 29	28	27 0
0	ESP	0

Table 5-10 *PageGrain* Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>0</td><td>31:29</td><td>Reserved. Must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
<tr><td>ESP</td><td>28</td><td>Enables support for 1KB pages.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>1KB page support is not enabled</td></tr><tr><td>1</td><td>1KB page support is enabled</td></tr></table>If this bit is a 1, the following changes occur to coprocessor 0 registers:
• The PFN field of the EntryLo0 and EntryLo1 registers holds the physical address down to bit 10 (the field is shifted left by 2 bits from the Release 1 definition)
• The MaskX field of the PageMask register is writable and is concatenated to the right of the Mask field to form the “don’t care” mask for the TLB entry.
• The VPN2X field of the EntryHi register is writable and bits 12..11 of the virtual address.
• The virtual address translation algorithm is modified to reflect the smaller page size.
If $Config3_{SP}$ = 0, 1KB pages are not implemented, and this bit is ignored on write and returns zero on read.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>27:0</td><td>Must be written as zero; returns zero on reads.</td><td>0</td><td>0</td></tr>
</table>

5.2.7 *Wired* Register (CP0 Register 6, Select 0)

The *Wired* register is a read/write register that specifies the boundary between the wired and random entries in the TLB as shown in Figure 5-7 on page 99. The width of the Wired field is calculated in the same manner as that described for the *Index* register above. Wired entries are fixed, non-replaceable entries that are not overwritten by a TLBWR instruction. Wired entries can be overwritten by a TLBWI instruction.

The *Wired* register is reset to zero by a Reset exception. Writing the *Wired* register causes the *Random* register to reset to its upper bound.

The operation of the processor is undefined if a value greater than or equal to the number of TLB entries is written to the *Wired* register.

This register is only valid with a TLB (4KEc core). It is reserved if the FM is implemented (4KEm and 4KEp cores).

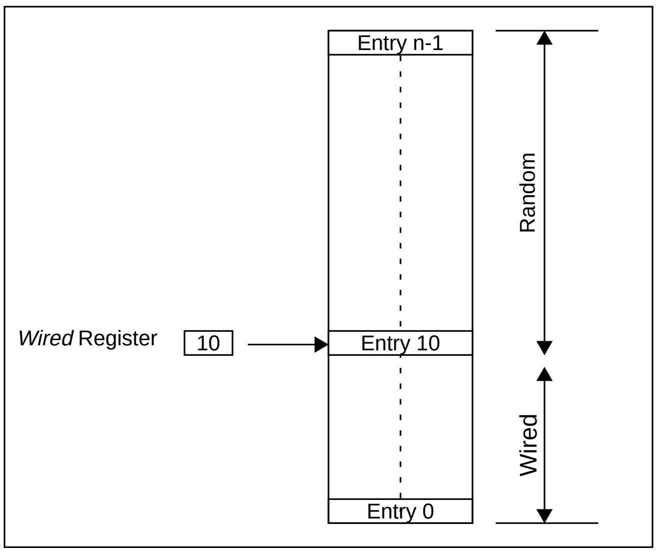


Figure 5-7 Wired and Random Entries in the TLB

Figure 5-8 *Wired* Register Format

31 … 4	3 … 0
0	Wired

Table 5-11 Wired Register Field Descriptions

Fields				
Name	**Bit(s)**	**Description**	**Read/ Write**	**Reset State**
0	31:4	Must be written as zero; returns zero on reads.	0	0
Wired	3:0	TLB wired boundary.	R/W	0

5.2.8 *HWREna* Register (CP0 Register 7, Select 0)

The *HWREna* register contains a bit mask that determines which hardware registers are accessible via the RDHWR instruction.

Figure 5-9 shows the format of the *HWREna* Register; Table 5-12 describes the *HWREna* register fields.

Figure 5-9 HWREna Register Format

31 ... 4	3 ... 0
0 0000 0000 0000 0000 0000 0000 0000	Mask

Table 5-12 HWREna Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read/ Write**	**Reset State**
0	31..4	Must be written with zero; returns zero on read	0	0
Mask	3..0	Each bit in this field enables access by the RDHWR instruction to a particular hardware register (which may not be an actual register). If bit 'n' in this field is a 1, access is enabled to hardware register 'n'. If bit 'n' of this field is a 0, access is disabled. See the RDHWR instruction for a list of valid hardware registers.	R/W	0

Privileged software may determine which of the hardware registers are accessible by the RDHWR instruction. In doing so, a register may be virtualized at the cost of handling a Reserved Instruction Exception, interpreting the instruction, and returning the virtualized value. For example, if it is not desirable to provide direct access to the *Count* register, access to that register may be individually disabled and the return value can be virtualized by the operating system.

5.2.9 *BadVAddr* Register (CP0 Register 8, Select 0)

The *BadVAddr* register is a read-only register that captures the most recent virtual address that caused one of the following exceptions:

- Address error (AdEL or AdES)
- TLB Refill (4KEc core)
- TLB Invalid (4KEc core)
- TLB Modified (4KEc core)

The *BadVAddr* register does not capture address information for cache or bus errors, since they are not addressing errors.

Figure 5-10 *BadVAddr* Register Format

31 ... 0
BadVAddr

Table 5-13 *BadVAddr* Register Field Description

Fields				
Name	**Bits**	**Description**	**Read/ Write**	**Reset State**
BadVAddr	31:0	Bad virtual address.	R	Undefined

5.2.10 *Count* Register (CP0 Register 9, Select 0)

The *Count* register acts as a timer, incrementing at a constant rate, whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. The counter increments every other clock, if the DC bit in the *Cause* register is 0.

The *Count* register can be written for functional or diagnostic purposes, including at reset or to synchronize processors.

By writing the CountDM bit in the *Debug* register, it is possible to control whether the *Count* register continues incrementing while the processor is in debug mode.

Figure 5-11 *Count* Register Format

31 … 0
Count

Table 5-14 *Count* Register Field Description

Fields				
Name	**Bits**	**Description**	**Read/ Write**	**Reset State**
Count	31:0	Interval counter.	R/W	Undefined

5.2.11 *EntryHi* Register (CP0 Register 10, Select 0)

The *EntryHi* register contains the virtual address match information used for TLB read, write, and access operations.

A TLB exception (TLB Refill, TLB Invalid, or TLB Modified) causes bits $VA_{31..13}$ of the virtual address to be written into the VPN2 field of the *EntryHi* register. An implementation of Release 2 of the Architecture which supports 1KB pages also writes $VA_{12..11}$ into the VPN2X field of the *EntryHi* register. A TLBR instruction writes the *EntryHi* register with the corresponding fields from the selected TLB entry. The ASID field is written by software with the current address space identifier value and is used during the TLB comparison process to determine TLB match.

Because the ASID field is overwritten by a TLBR instruction, software must save and restore the value of ASID around use of the TLBR. This is especially important in TLB Invalid and TLB Modified exceptions, and in other memory management software.

The VPNX2 and VPN2 fields of the *EntryHi* register are not defined after an address error exception and these fields may be modified by hardware during the address error exception sequence. Software writes of the *EntryHi* register (via MTC0) do not cause the implicit write of address-related fields in the *BadVAddr*, *Context* registers.

This register is only valid with the TLB (4KEc core). It is reserved if the FM is implemented (4KEm and 4KEp cores).

Figure 5-12 *EntryHi* Register Format

31 ... 13	12 ... 11	10 ... 8	7 ... 0
VPN2	VPN2X	0	ASID

Table 5-15 *EntryHi* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
VPN2	31..13	$VA_{31..13}$ of the virtual address (virtual page number / 2). This field is written by hardware on a TLB exception or on a TLB read, and is written by software before a TLB write.	R/W	Undefined
VPN2X	12..11	In Release 2 of the Architecture, the VPN2X field is an extension to the VPN2 field to support 1KB pages. These bits are not writable by either hardware or software unless $Config3_{SP}$ = 1 and $PageGrain_{ESP}$ = 1. If enabled for write, this field contains $VA_{12..11}$ of the virtual address and is written by hardware on a TLB exception or on a TLB read, and is by software before a TLB write. If writes are not enabled, and in implementations of Release 1 of the Architecture, this field must be written with zero and returns zeros on read.	R/W	0
0	10..8	Must be written as zero; returns zero on read.	0	0
ASID	7..0	Address space identifier. This field is written by hardware on a TLB read and by software to establish the current ASID value for TLB write and against which TLB references match each entry's TLB ASID field.	R/W	Undefined

5.2.12 *Compare* Register (CP0 Register 11, Select 0)

The *Compare* register acts in conjunction with the *Count* register to implement a timer and timer interrupt function. The timer interrupt is an output of the cores. The *Compare* register maintains a stable value and does not change on its own.

When the value of the *Count* register equals the value of the *Compare* register, the SI_TimerInt pin is asserted. This pin will remain asserted until the *Compare* register is written. The SI_TimerInt pin can be fed back into the core on one of the interrupt pins to generate an interrupt. Traditionally, this has been done by multiplexing it with hardware interrupt 5 to set interrupt bit IP(7) in the *Cause* register.

For diagnostic purposes, the *Compare* register is a read/write register. In normal use, however, the *Compare* register is write-only. Writing a value to the *Compare* register, as a side effect, clears the timer interrupt.

Figure 5-13 *Compare* Register Format

31 … 0
Compare

Table 5-16 *Compare* Register Field Description

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
Compare	31:0	Interval count compare value.	R/W	Undefined

5.2.13 *Status* Register (CP0 Register 12, Select 0)

The *Status* register is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. Fields of this register combine to create operating modes for the processor. Refer to <<NEED CROSSREF>> for a discussion of operating modes, and Section <<NEED CROSSREF>> for a discussion of interrupt modes.

Interrupt Enable: Interrupts are enabled when all of the following conditions are true:

- IE = 1
- EXL = 0
- ERL = 0
- DM = 0

If these conditions are met, then the settings of the IM and IE bits enable the interrupts.

Operating Modes: If the DM bit in the Debug register is 1, then the processor is in debug mode; otherwise the processor is in either kernel or user mode. The following CPU Status register bit settings determine user or kernel mode:

- User mode: UM = 1, EXL = 0, and ERL = 0
- Kernel mode: UM = 0, or EXL = 1, or ERL = 1

Coprocessor Accessibility: The *Status* register CU bits control coprocessor accessibility. If any coprocessor is unusable, then an instruction that accesses it generates an exception.

Figure 5-14 shows the format of the *Status* register; Table 5-17 describes the *Status* register fields.

Figure 5-14 Status Register Format

31 28	27	26	25	24 23	22	21	20	19	18	17 16	15 10	9 8	7 6 5	4	3	2	1	0
CU3..CU0	RP	FR	RE	R	BEV	TS	SR	NMI	0	R	IM7..IM2	IM1..IM0	R	UM	R	ERL	EXL	IE
											IPL							

Table 5-17 Status Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
CU3	31	Controls access to coprocessor 3. COP3 is not supported. This bit cannot be written and will read as 0.	R	0
CU2	30	Controls access to coprocessor 2. This bit can only be written if coprocessor is attached to the COP2 interface. (C2 bit in Config1 is set). This bit will read as 0 if no coprocessor is present.	R/W	0
CU1	29	Controls access to Coprocessor 1. COP1 is not supported. This bit cannot be written and will read as 0.	R	0

Table 5-17 Status Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>CU0</td><td>28</td><td>Controls access to coprocessor 0

0: access not allowed
1: access allowed

Coprocessor 0 is always usable when the processor is running in kernel mode, independent of the state of the CU0 bit.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>RP</td><td>27</td><td>Enables reduced power mode. The state of the RP bit is available on the external core interface as the SI_RP signal.</td><td>R/W</td><td>0 for Cold Reset only.</td></tr>
<tr><td>FR</td><td>26</td><td>This bit is related to floating point registers. Since the 4KE core does not contain a floating point unit, this bit is ignored on write and read as zero.</td><td>R</td><td>0</td></tr>
<tr><td>RE</td><td>25</td><td>Used to enable reverse-endian memory references while the processor is running in user mode:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>User mode uses configured endianness</td></tr><tr><td>1</td><td>User mode uses reversed endianness</td></tr></table>
Neither Debug Mode nor Kernel Mode nor Supervisor Mode references are affected by the state of this bit.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>R</td><td>24:23</td><td>Reserved. This field is ignored on write and read as 0.</td><td>R</td><td>0</td></tr>
<tr><td>BEV</td><td>22</td><td>Controls the location of exception vectors:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Normal</td></tr><tr><td>1</td><td>Bootstrap</td></tr></table></td><td>R/W</td><td>1</td></tr>
<tr><td>TS</td><td>21</td><td>TLB shutdown. Indicates that the TLB has detected a match on multiple entries. This bit is set if a TLBWI or TLBWR instruction is issued that would cause a TLB shutdown condition if allowed to complete. A machine check exception is also issued. This bit is only used in the 4KEc processor and is reserved in the 4KEp and 4KEm processors.

Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition.</td><td>R/W</td><td>0</td></tr>
<tr><td>SR</td><td>20</td><td>Indicates that the entry through the reset exception vector was due to a Soft Reset:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Not Soft Reset (NMI or Reset)</td></tr><tr><td>1</td><td>Soft Reset</td></tr></table>
Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition.</td><td>R/W</td><td>1 for Soft Reset; 0 otherwise</td></tr>
</table>

Table 5-17 Status Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>NMI</td><td>19</td><td>Indicates that the entry through the reset exception vector was due to an NMI:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Not NMI (Soft Reset or Reset)</td></tr><tr><td>1</td><td>NMI</td></tr></table>
Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition.</td><td>R/W</td><td>1 for NMI; 0 otherwise</td></tr>
<tr><td>0</td><td>18</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
<tr><td>R</td><td>17:16</td><td>Reserved. Ignored on write and read as zero.</td><td>R</td><td>0</td></tr>
<tr><td>IM7..IM2</td><td>15..10</td><td>Interrupt Mask: Controls the enabling of each of the hardware interrupts. Refer to Section 4.3, "Interrupts" on page 55 for a complete discussion of enabled interrupts.

An interrupt is taken if interrupts are enabled and the corresponding bits are set in both the Interrupt Mask field of the Status register and the Interrupt Pending field of the Cause register and the IE bit is set in the Status register.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupt request disabled</td></tr><tr><td>1</td><td>Interrupt request enabled</td></tr></table>
In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled ($Config3_{VEIC}$ = 1), these bits take on a different meaning and are interpreted as the IPL field, described below.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>IPL</td><td>15..10</td><td>Interrupt Priority Level.

In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled ($Config3_{VEIC}$ = 1), this field is the encoded (0..63) value of the current IPL. An interrupt will be signaled only if the requested IPL is higher than this value.

If EIC interrupt mode is not enabled ($Config3_{VEIC}$ = 0), these bits take on a different meaning and are interpreted as the IM7..IM2 bits, described above.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>IM1..IM0</td><td>9..8</td><td>Interrupt Mask: Controls the enabling of each of the software interrupts. Refer to Section <<NEED CROSSREF>> for a complete discussion of enabled interrupts.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupt request disabled</td></tr><tr><td>1</td><td>Interrupt request enabled</td></tr></table>
In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled ($Config3_{VEIC}$ = 1), these bits are writable, but have no effect on the interrupt system.</td><td>R/W</td><td>Undefined</td></tr>
</table>

Table 5-17 Status Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>R</td><td>7:5</td><td>Reserved. This field is ignored on write and read as 0.</td><td>R</td><td>0</td></tr>
<tr><td>UM</td><td>4</td><td>This bit denotes the base operating mode of the processor. See Section 3.2, "Modes of Operation" on page 34 for a full discussion of operating modes. The encoding of this bit is:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Base mode is Kernel Mode</td></tr><tr><td>1</td><td>Base mode is User Mode</td></tr></table>
Note that the processor can also be in kernel mode if ERL or EXL is set, regardless of the state of the UM bit.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>R</td><td>3</td><td>This bit is reserved. This bit is ignored on write and read as zero.</td><td>R</td><td>0</td></tr>
<tr><td>ERL</td><td>2</td><td>Error Level; Set by the processor when a Reset, Soft Reset, NMI or Cache Error exception are taken.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Normal level</td></tr><tr><td>1</td><td>Error level</td></tr></table>
When ERL is set:
• The processor is running in kernel mode
• Interrupts are disabled
• The ERET instruction will use the return address held in ErrorEPC instead of EPC
• The lower 2^{29} bytes of kuseg are treated as an unmapped and uncached region. See Chapter 3, “Memory Management,” on page 34. This allows main memory to be accessed in the presence of cache errors. The operation of the processor is UNDEFINED if the ERL bit is set while the processor is executing instructions from kuseg.</td><td>R/W</td><td>1</td></tr>
<tr><td>EXL</td><td>1</td><td>Exception Level; Set by the processor when any exception other than Reset, Soft Reset, or NMI exceptions is taken.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Normal level</td></tr><tr><td>1</td><td>Exception level</td></tr></table>
When EXL is set:
• The processor is running in Kernel Mode
• Interrupts are disabled.
• TLB Refill exceptions use the general exception vector instead of the TLB Refill vector.
• EPC, $Cause_{BD}$ and SRSCtl (implementations of Release 2 of the Architecture only) will not be updated if another exception is taken</td><td>R/W</td><td>Undefined</td></tr>
</table>

Table 5-17 Status Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IE</td><td>0</td><td>Interrupt Enable: Acts as the master enable for software and hardware interrupts:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupts are disabled</td></tr><tr><td>1</td><td>Interrupts are enabled</td></tr></table>
In Release 2 of the Architecture, this bit may be modified separately via the DI and EI instructions.</td><td>R/W</td><td>Undefined</td></tr>
</table>

5.2.14 *IntCtl* Register (CP0 Register 12, Select 1)

The *IntCtl* register controls the expanded interrupt capability added in Release 2 of the Architecture, including vectored interrupts and support for an external interrupt controller. This register does not exist in implementations of Release 1 of the Architecture.

Figure 5-15 shows the format of the *IntCtl* register; Table 5-18 describes the *IntCtl* register fields.

Figure 5-15 IntCtl Register Format

31 29	28 26	25 10	9 5	4 0
IPTI	IPPCI	0	VS	0

Table 5-18 IntCtl Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IPTI</td><td>31..29</td><td>For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Timer Interrupt request is merged, and allows software to determine whether to consider $Cause_{TI}$ for a potential interrupt.
<table><tr><th>Encoding</th><th>IP bit</th><th>Hardware Interrupt Source</th></tr><tr><td>2</td><td>2</td><td>HW0</td></tr><tr><td>3</td><td>3</td><td>HW1</td></tr><tr><td>4</td><td>4</td><td>HW2</td></tr><tr><td>5</td><td>5</td><td>HW3</td></tr><tr><td>6</td><td>6</td><td>HW4</td></tr><tr><td>7</td><td>7</td><td>HW5</td></tr></table>
The value of this bit is set by the static input, *SI_IPTI[2:0]*. This allows external logic to communicate the specific *SI_Int* hardware interrupt pin to which the *SI_TimerInt* signal is attached.

The value of this field is not meaningful if External Interrupt Controller Mode is enabled. The external interrupt controller is expected to provide this information for that interrupt mode.</td><td>R</td><td>Externally Set</td></tr>
<tr><td>IPPCI</td><td>28..26</td><td>For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Performance Counter Interrupt request is merged, and allows software to determine whether to consider $Cause_{PCI}$ for a potential interrupt.

Since performance counters are not implemented on the 4KE core ($Config1_{PC}$ = 0), this field is ignored on write and returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>0</td><td>25..10</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
</table>

Table 5-18 IntCtl Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>VS</td><td>9..5</td><td>Vector Spacing. If vectored interrupts are implemented (as denoted by Config3$_{VInt}$ or Config3$_{VEIC}$), this field specifies the spacing between vectored interrupts.
<table>
<tr><th>Encoding</th><th>Spacing Between Vectors (hex)</th><th>Spacing Between Vectors (decimal)</th></tr>
<tr><td>16#00</td><td>16#000</td><td>0</td></tr>
<tr><td>16#01</td><td>16#020</td><td>32</td></tr>
<tr><td>16#02</td><td>16#040</td><td>64</td></tr>
<tr><td>16#04</td><td>16#080</td><td>128</td></tr>
<tr><td>16#08</td><td>16#100</td><td>256</td></tr>
<tr><td>16#10</td><td>16#200</td><td>512</td></tr>
</table>

All other values are reserved. The operation of the processor is UNDEFINED if a reserved value is written to this field.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>4..0</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
</table>

5.2.15 *SRSCtl* Register (CP0 Register 12, Select 2)

The *SRSCtl* register controls the operation of GPR shadow sets in the processor. This register does not exist in implementations of the architecture prior to Release 2.

Figure 5-16 shows the format of the *SRSCtl* register; Table 5-19 describes the *SRSCtl* register fields.

Figure 5-16 SRSCtl Register Format

31 30	29 26	25 22	21 18	17 16	15 12	11 10	9 6	5 4	3 0
0 00	HSS	0 00 00	EICSS	0 00	ESS	0 00	PSS	0 00	CSS

Table 5-19 SRSCtl Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>0</td><td>31..30</td><td>Must be written as zeros; returns zero on read.</td><td>0</td><td>0</td></tr>
<tr><td>HSS</td><td>29..26</td><td>Highest Shadow Set. This field contains the highest shadow set number that is implemented by this processor. A value of zero in this field indicates that only the normal GPRs are implemented.

Possible values of this field for the 4KE processor are:

<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>One shadow set (normal GPR set) is present.</td></tr><tr><td>1</td><td>Two shadow sets are present.</td></tr><tr><td>3</td><td>Four shadow sets are present.</td></tr><tr><td>2, 3-15</td><td>Reserved</td></tr></table>
The value in this field also represents the highest value that can be written to the ESS, EICSS, PSS, and CSS fields of this register, or to any of the fields of the SRSMap register. The operation of the processor is UNDEFINED if a value larger than the one in this field is written to any of these other fields.</td><td>R</td><td>Preset</td></tr>
<tr><td>0</td><td>25..22</td><td>Must be written as zeros; returns zero on read.</td><td>0</td><td>0</td></tr>
<tr><td>EICSS</td><td>21..18</td><td>EIC interrupt mode shadow set. If $\text{Config3}_{\text{VEIC}}$ is 1 (EIC interrupt mode is enabled), this field is loaded from the external interrupt controller for each interrupt request and is used in place of the SRSMap register to select the current shadow set for the interrupt.

See Section 4.3.1.3, "External Interrupt Controller Mode" on page 60 for a discussion of EIC interrupt mode. If $\text{Config3}_{\text{VEIC}}$ is 0, this field must be written as zero, and returns zero on read.</td><td>R</td><td>Undefined</td></tr>
<tr><td>0</td><td>17..16</td><td>Must be written as zeros; returns zero on read.</td><td>0</td><td>0</td></tr>
</table>

Table 5-19 SRSCtl Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
ESS	15..12	Exception Shadow Set. This field specifies the shadow set to use on entry to Kernel Mode caused by any exception other than a vectored interrupt. The operation of the processor is **UNDEFINED** if software writes a value into this field that is greater than the value in the HSS field.	R/W	0
0	11..10	Must be written as zeros; returns zero on read.	0	0
PSS	9..6	Previous Shadow Set. If GPR shadow registers are implemented, and with the exclusions noted in the next paragraph, this field is copied from the CSS field when an exception or interrupt occurs. An ERET instruction copies this value back into the CSS field if $Status_{BEV}$ = 0. This field is not updated on any exception which sets $Status_{ERL}$ to 1 (i.e., Reset, Soft Reset, NMI, cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with $Status_{EXL}$ = 1, or $Status_{BEV}$ = 1. This field is not updated on an exception that occurs while $Status_{ERL}$ = 1. The operation of the processor is **UNDEFINED** if software writes a value into this field that is greater than the value in the HSS field.	R/W	0
0	5..4	Must be written as zeros; returns zero on read.	0	0
CSS	3..0	Current Shadow Set. If GPR shadow registers are implemented, this field is the number of the current GPR set. With the exclusions noted in the next paragraph, this field is updated with a new value on any interrupt or exception, and restored from the PSS field on an ERET. Table 5-20 describes the various sources from which the CSS field is updated on an exception or interrupt. This field is not updated on any exception which sets $Status_{ERL}$ to 1 (i.e., Reset, Soft Reset, NMI, cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with $Status_{EXL}$ = 1, or $Status_{BEV}$ = 1. Neither is it updated on an ERET with $Status_{ERL}$ = 1 or $Status_{BEV}$ = 1. This field is not updated on an exception that occurs while $Status_{ERL}$ = 1. The value of CSS can be changed directly by software only by writing the PSS field and executing an ERET instruction.	R	0

Table 5-20 Sources for new $SRSCtl_{CSS}$ on an Exception or Interrupt

Exception Type	Condition	$SRSCtl_{CSS}$ Source	Comment
Exception	All	$SRSCtl_{ESS}$	
Non-Vectored Interrupt	$Cause_{IV}$ = 0	$SRSCtl_{ESS}$	Treat as exception

Table 5-20 Sources for new $SRSCtl_{CSS}$ on an Exception or Interrupt

Exception Type	Condition	$SRSCtl_{CSS}$ Source	Comment
Vectored Interrupt	$Cause_{IV}$ = 1 and $Config3_{VEIC}$ = 0 and $Config3_{VInt}$ = 1	$SRSMap_{\mathtt{VECTNUM}}$	Source is internal map register. (for `VECTNUM` see Table 4-3)
Vectored EIC Interrupt	$Cause_{IV}$ = 1 and $Config3_{VEIC}$ = 1	$SRSCtl_{EICSS}$	Source is external interrupt controller.

5.2.16 *SRSMap* Register (CP0 Register 12, Select 3)

The *SRSMap* register contains 8 4-bit fields that provide the mapping from an vector number to the shadow set number to use when servicing such an interrupt. The values from this register are not used for a non-interrupt exception, or a non-vectored interrupt ($Cause_{IV}$ = 0 or $IntCtl_{VS}$ = 0). In such cases, the shadow set number comes from $SRSCtl_{ESS}$.

If $SRSCtl_{HSS}$ is zero, the results of a software read or write of this register are **UNPREDICTABLE**.

The operation of the processor is **UNDEFINED** if a value is written to any field in this register that is greater than the value of $SRSCtl_{HSS}$.

The *SRSMap* register contains the shadow register set numbers for vector numbers 7..0. The same shadow set number can be established for multiple interrupt vectors, creating a many-to-one mapping from a vector to a single shadow register set number.

Figure 5-17 shows the format of the *SRSMap* register; Table 5-21 describes the *SRSMap* register fields.

Figure 5-17 SRSMap Register Format

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
SSV7	SSV6	SSV5	SSV4	SSV3	SSV2	SSV1	SSV0

Table 5-21 SRSMap Register Field Descriptions

Fields				
Name	Bits	Description	Read/ Write	Reset State
SSV7	31..28	Shadow register set number for Vector Number 7	R/W	0
SSV6	27..24	Shadow register set number for Vector Number 6	R/W	0
SSV5	23..20	Shadow register set number for Vector Number 5	R/W	0
SSV4	19..16	Shadow register set number for Vector Number 4	R/W	0
SSV3	15..12	Shadow register set number for Vector Number 3	R/W	0
SSV2	11..8	Shadow register set number for Vector Number 2	R/W	0
SSV1	7..4	Shadow register set number for Vector Number 1	R/W	0
SSV0	3..0	Shadow register set number for Vector Number 0	R/W	0

5.2.17 *Cause* Register (CP0 Register 13, Select 0)

The *Cause* register primarily describes the cause of the most recent exception. In addition, fields also control software interrupt requests and the vector through which interrupts are dispatched. With the exception of the $IP_{1..0}$, DC, IV, and WP fields, all fields in the *Cause* register are read-only. Release 2 of the Architecture added optional support for an External Interrupt Controller (EIC) interrupt mode, in which $IP_{7..2}$ are interpreted as the Requested Interrupt Priority Level (RIPL).

Figure 5-18 shows the format of the *Cause* register; Table 5-22 describes the *Cause* register fields.

Figure 5-18 Cause Register Format

31	30	29 28	27	26	25 24	23	22	21 16	15 10	9 8	7	6 2	1 0
BD	TI	CE	DC	PCI	0	IV	WP	0	IP7..IP2	IP1..IP0	0	Exc Code	0
									RIPL				

Table 5-22 Cause Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>BD</td><td>31</td><td>Indicates whether the last exception taken occurred in a branch delay slot:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Not in delay slot</td></tr>
<tr><td>1</td><td>In delay slot</td></tr>
</table>
The processor updates BD only if $Status_{EXL}$ was zero when the exception occurred.</td><td>R</td><td>Undefined</td></tr>
<tr><td>TI</td><td>30</td><td>Timer Interrupt. This bit denotes whether a timer interrupt is pending (analogous to the IP bits for other interrupt types):
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No timer interrupt is pending</td></tr>
<tr><td>1</td><td>Timer interrupt is pending</td></tr>
</table>
The state of the TI bit is available on the external core interface as the SI_TimerInt signal</td><td>R</td><td>Undefined</td></tr>
<tr><td>CE</td><td>29..28</td><td>Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. This field is loaded by hardware on every exception, but is UNPREDICTABLE for all exceptions except for Coprocessor Unusable.</td><td>R</td><td>Undefined</td></tr>
</table>

Table 5-22 Cause Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>DC</td><td>27</td><td>Disable Count register. In some power-sensitive applications, the Count register is not used and is the source of meaningful power dissipation. This bit allows the Count register to be stopped in such situations.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Enable counting of Count register</td></tr><tr><td>1</td><td>Disable counting of Count register</td></tr></table></td><td>R/W</td><td>0</td></tr>
<tr><td>PCI</td><td>26</td><td>Performance Counter Interrupt. In an implementation of Release 2 of the Architecture, this bit denotes whether a performance counter interrupt is pending (analogous to the IP bits for other interrupt types):
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No timer interrupt is pending</td></tr><tr><td>1</td><td>Timer interrupt is pending</td></tr></table>
Since performance counters are not implemented ($Config1_{PC} = 0$), this bit must be written as zero and returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>IV</td><td>23</td><td>Indicates whether an interrupt exception uses the general exception vector or a special interrupt vector:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Use the general exception vector (16#180)</td></tr><tr><td>1</td><td>Use the special interrupt vector (16#200)</td></tr></table>
In implementations of Release 2 of the architecture, if the $Cause_{IV}$ is 1 and $Status_{BEV}$ is 0, the special interrupt vector represents the base of the vectored interrupt table.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>WP</td><td>22</td><td>Indicates that a watch exception was deferred because $Status_{EXL}$ or $Status_{ERL}$ were a one at the time the watch exception was detected. This bit both indicates that the watch exception was deferred, and causes the exception to be initiated once $Status_{EXL}$ and $Status_{ERL}$ are both zero. As such, software must clear this bit as part of the watch exception handler to prevent a watch exception loop.

Software should not write a 1 to this bit when its value is a 0, thereby causing a 0-to-1 transition. If such a transition is caused by software, it is UNPREDICTABLE whether hardware ignores the write, accepts the write with no side effects, or accepts the write and initiates a watch exception once $Status_{EXL}$ and $Status_{ERL}$ are both zero.</td><td>R/W</td><td>Undefined</td></tr>
</table>

Table 5-22 Cause Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IP7..IP2</td><td>15..10</td><td>Indicates an interrupt is pending:
<table><tr><th>Bit</th><th>Name</th><th>Meaning</th></tr><tr><td>15</td><td>IP7</td><td>Hardware interrupt 5</td></tr><tr><td>14</td><td>IP6</td><td>Hardware interrupt 4</td></tr><tr><td>13</td><td>IP5</td><td>Hardware interrupt 3</td></tr><tr><td>12</td><td>IP4</td><td>Hardware interrupt 2</td></tr><tr><td>11</td><td>IP3</td><td>Hardware interrupt 1</td></tr><tr><td>10</td><td>IP2</td><td>Hardware interrupt 0</td></tr></table>
If EIC interrupt mode is not enabled ($Config3_{VEIC}$ = 0), timer interrupts are combined in a system-dependent way with any hardware interrupt. If EIC interrupt mode is enabled ($Config3_{VEIC}$ = 1), these bits take on a different meaning and are interpreted as the RIPL field, described below.

See Section 4.3, "Interrupts" on page 55 for a general description of interrupt processing.</td><td>R</td><td>Undefined</td></tr>
<tr><td>RIPL</td><td>15..10</td><td>Requested Interrupt Priority Level.

If EIC interrupt mode is enabled ($Config3_{VEIC}$ = 1), this field is the encoded (0..63) value of the requested interrupt. A value of zero indicates that no interrupt is requested.

If EIC interrupt mode is not enabled ($Config3_{VEIC}$ = 0), these bits take on a different meaning and are interpreted as the IP7..IP2 bits, described above.</td><td>R</td><td>Undefined</td></tr>
<tr><td>IP1..IP0</td><td>9..8</td><td>Controls the request for software interrupts:
<table><tr><th>Bit</th><th>Name</th><th>Meaning</th></tr><tr><td>9</td><td>IP1</td><td>Request software interrupt 1</td></tr><tr><td>8</td><td>IP0</td><td>Request software interrupt 0</td></tr></table>
These bits are exported to an external interrupt controller for prioritization in EIC interrupt mode with other interrupt sources. The state of these bits is available on the external core interface as the *SI_SWInt[1:0]* bus.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>ExcCode</td><td>6..2</td><td>Exception code - see Table 5-23</td><td>R</td><td>Undefined</td></tr>
<tr><td>0</td><td>25..24, 21..16, 7, 1..0</td><td>Must be written as zero; returns zero on read.</td><td>0</td><td>0</td></tr>
</table>

Table 5-23 Cause Register ExcCode Field

Exception Code Value			
Decimal	**Hexadecimal**	**Mnemonic**	**Description**
0	16#00	Int	Interrupt
1	16#01	Mod	TLB modification exception (4KEc core)
2	16#02	TLBL	TLB exception (load or instruction fetch) (4KEc core)
3	16#03	TLBS	TLB exception (store) (4KEc core)
4	16#04	AdEL	Address error exception (load or instruction fetch)
5	16#05	AdES	Address error exception (store)
6	16#06	IBE	Bus error exception (instruction fetch)
7	16#07	DBE	Bus error exception (data reference: load or store)
8	16#08	Sys	Syscall exception
9	16#09	Bp	Breakpoint exception
10	16#0a	RI	Reserved instruction exception
11	16#0b	CpU	Coprocessor Unusable exception
12	16#0c	Ov	Arithmetic Overflow exception
13	16#0d	Tr	Trap exception
14-15	16#0e-16#0f	-	Reserved
16	16#10	IS1	Implementation-Specific Exception 1 (COP2)
17	16#11	IS2	Implementation-Specific Exception 2(COP2)
18	16#12	C2E	Coprocessor 2 exceptions
19-22	16#13-16#16	-	Reserved
23	16#17	WATCH	Reference to WatchHi/WatchLo address
24	16#18	MCheck	Machine check
25-31	16#19-16#1f	-	Reserved

5.2.18 Exception Program Counter (CP0 Register 14, Select 0)

The Exception Program Counter (*EPC*) is a read/write register that contains the address at which processing resumes after an exception has been serviced. All bits of the *EPC* register are significant and must be writable.

For synchronous (precise) exceptions, the *EPC* contains one of the following:

- The virtual address of the instruction that was the direct cause of the exception
- The virtual address of the immediately preceding branch or jump instruction, when the exception causing instruction is in a branch delay slot and the *Branch Delay* bit in the *Cause* register is set.

On new exceptions, the processor does not write to the *EPC* register when the EXL bit in the *Status* register is set, however, the register can still be written via the MTC0 instruction.

In processors that implement the MIPS16 ASE, a read of the EPC register (via MFC0) returns the following value in the destination GPR:

$$\mathtt{GPR[rt]} \leftarrow \mathtt{ExceptionPC}_{31..1} \;||\; \mathtt{ISAMode}_0$$

That is, the upper 31 bits of the exception PC are combined with the lower bit of the ISAMode field and written to the GPR.

Similarly, a write to the EPC register (via MTC0) takes the value from the GPR and distributes that value to the exception PC and the ISAMode field, as follows

$$\mathtt{ExceptionPC} \leftarrow \mathtt{GPR[rt]}_{31..1} \;||\; 0$$
$$\mathtt{ISAMode} \leftarrow 2\#0 \;||\; \mathtt{GPR[rt]}_0$$

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the exception PC, and the lower bit of the exception PC is cleared. The upper bit of the ISAMode field is cleared and the lower bit is loaded from the lower bit of the GPR.

Figure 5-19 *EPC* Register Format

31 ... 0
EPC

Table 5-24 *EPC* Register Field Description

Fields				
Name	Bit(s)	Description	Read/ Write	Reset State
EPC	31:0	Exception Program Counter.	R/W	Undefined

5.2.19 Processor Identification (CP0 Register 15, Select 0)

The Processor Identification (*PRId*) register is a 32 bit read-only register that contains information identifying the manufacturer, manufacturer options, processor identification, and revision level of the processor.

Figure 5-20 *PRId* Register Format

31 ... 24	23 ... 16	15 ... 8	7 ... 5 / 4 ... 2 / 1 0
R	Company ID	Processor ID	Revision

Table 5-25 *PRId* Register Field Descriptions

Fields				
Name	**Bit(s)**	**Description**	**Read/ Write**	**Reset State**
R	31:24	Reserved. Must be ignored on write and read as zero	R	0
Company ID	23:16	Identifies the company that designed or manufactured the processor. In the 4KE this field contains a value of 1 to indicate MIPS Technologies, Inc.	R	1
Processor ID	15:8	Identifies the type of processor. This field allows software to distinguish between the various types of MIPS Technologies processors.	R	4KEc core - 0x90 4KEm & 4KEp cores - 0x91
Revision	7:0	Specifies the revision number of the processor. This field allows software to distinguish between one revision and another of the same processor type. This field is broken up into the following three subfields	R	Preset
Major Revision	7:5	This number is increased on major revisions of the processor core	R	Preset
Minor Revision	4:2	This number is increased on each incremental revision of the processor and reset on each new major revision	R	Preset
Patch Level	1:0	If a patch is made to modify an older revision of the processor, this field will be incremented	R	Preset

5.2.20 *EBase* Register (CP0 Register 15, Select 1)

The *EBase* register is a read/write register containing the base address of the exception vectors used when $Status_{BEV}$ equals 0, and a read-only CPU number value that may be used by software to distinguish different processors in a multi-processor system.

The *EBase* register provides the ability for software to identify the specific processor within a multi-processor system, and allows the exception vectors for each processor to be different, especially in systems composed of heterogeneous processors. Bits 31..12 of the *EBase* register are concatenated with zeros to form the base of the exception vectors when $Status_{BEV}$ is 0. The exception vector base address comes from the fixed defaults (see Section 4.5, "Exception Vector Locations" on page 65) when $Status_{BEV}$ is 1, or for any EJTAG Debug exception. The reset state of bits 31..12 of the *EBase* register initialize the exception base register to `16#8000.0000`, providing backward compatibility with Release 1 implementations.

Bits 31..30 of the *EBase* Register are fixed with the value `2#10` to force the exception base address to be in the kseg0 or kseg1 unmapped virtual address segments.

If the value of the exception base register is to be changed, this must be done with $Status_{BEV}$ equal 1. The operation of the processor is **UNDEFINED** if the Exception Base field is written with a different value when $Status_{BEV}$ is 0.

Combining bits 31..20 with the Exception Base field allows the base address of the exception vectors to be placed at any 4KBbyte page boundary.

Figure 5-21 shows the format of the *EBase* Register; Table 5-26 describes the *EBase* register fields.

Figure 5-21 EBase Register Format

31	30	29 .. 12	11 10	9 .. 0
1	0	Exception Base	0 0	CPUNum

Table 5-26 EBase Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read/ Write**	**Reset State**
1	31	This bit is ignored on write and returns one on read.	R	1
0	30	This bit is ignored on write and returns zero on read.	R	0
Exception Base	29..12	In conjunction with bits 31..30, this field specifies the base address of the exception vectors when $Status_{BEV}$ is zero.	R/W	0
0	11..10	Must be written as zero; returns zero on read.	0	0
CPUNum	9..0	This field specifies the number of the CPU in a multi-processor system and can be used by software to distinguish a particular processor from the others. The value in this field is set by the *SI_CPUNum[9:0]* static input pins to the core. In a single processor system, this value should be set to zero.	R	Externally Set

5.2.21 *Config* Register (CP0 Register 16, Select 0)

The *Config* register specifies various configuration and capabilities information. Most of the fields in the *Config* register are initialized by hardware during the Reset exception process, or are constant. The K0, KU, and K23 fields must be initialized by software in the Reset exception handler, if the reset value is not desired.

Figure 5-22 *Config* Register Format — Select 0

31	30 28	27 25	24	23	22	21	20	19	18 17	16	15	14 13	12 10	9 7	6 3	2 0
M	K23	KU	ISP	DSP	UDI	SB	MDU	0	MM	BM	BE	AT	AR	MT	0	K0

Figure 5-23 *Config* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
M	31	This bit is hardwired to '1' to indicate the presence of the Config1 register.	R	1
K23	30:28	This field controls the cacheability of the kseg2 and kseg3 address segments in FM implementations. This field is valid in the 4KEp and 4KEm processor and is reserved in the 4KEc processor. Refer to Table 5-27 for the field encoding.	FM: R/W TLB: R	FM: 010 TLB: 000
KU	27:25	This field controls the cacheability of the kuseg and useg address segments in FM implementations. This field is valid in the 4KEp and 4KEm processor and is reserved in the 4KEc processor. Refer to Table 5-27 for the field encoding.	FM: R/W TLB: R	FM: 010 TLB: 000
ISP	24	Indicates whether Instruction ScratchPad RAM is present. Set by the *ISP_Present* static input pin, if scratchpad was enabled when the core was built. 0 = No Instruction ScratchPad is present 1 = Instruction ScratchPad is present	R	Externally Set
DSP	23	Indicates whether Data ScratchPad RAM is present. Set by the *DSP_Present* static input pin, if scratchpad was enabled when the core was built. 0 = No Data ScratchPad is present 1 = Data ScratchPad is present	R	Externally Set
UDI	22	This bit indicates that CorExtend User Defined Instructions have been implemented. 0 = No User Defined Instructions are implemented 1 = User Defined Instructions are implemented	R	Preset
SB	21	Indicates whether SimpleBE bus mode is enabled. Set via *SI_SimpleBE[0]* input pin. 0 = No reserved byte enables on EC interface 1 = Only simple byte enables allowed on EC interface	R	Externally Set
MDU	20	This bit indicates the type of Multiply/Divide Unit present. 0 = Fast, high-performance MDU (4KEc and 4KEm cores) 1 = Iterative, area-efficient MDU (4KEp cores)	R	Preset

Figure 5-23 *Config* Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
0	19	Must be written as 0. Returns zero on reads.	0	0
MM	18:17	This bit indicates whether merging is enabled in the 32 byte collapsing write buffer. Set via *SI_MergeMode[1:0]* input pins: 00 = No Merging 10 = Merging allowed x1 = Reserved	R	Externally Set
BM	16	Burst order. Set via *EB_SBlock* input pin. 0: Sequential 1: SubBlock	R	Externally Set
BE	15	Indicates the endian mode in which the processor is running. Set via *SI_Endian* input pin. 0: Little endian 1: Big endian	R	Externally Set
AT	14:13	Architecture type implemented by the processor. This field is always 00 to indicate the MIPS32 architecture.	R	00
AR	12:10	Architecture revision level. This field is always 001 to indicate MIPS32 Release 2. 0: Release 1 1: Release 2 2-7: Reserved	R	001
MT	9:7	MMU Type: 1: Standard TLB (4KEc core) 3: Fixed Mapping(4KEp, 4KEm cores) 0, 2, 4-7: Reserved	R	Preset
0	6:3	Must be written as zeros; returns zeros on reads.	0	0
K0	2:0	Kseg0 coherency algorithm. Refer to Table 5-27 for the field encoding.	R/W	010

Table 5-27 Cache Coherency Attributes

C(2:0) Value	Cache Coherency Attribute
0	Cacheable, noncoherent, write-through, no write allocate
1	Cacheable, noncoherent, write-through, write allocate
3*, 4, 5, 6	Cacheable, noncoherent, write-back, write allocate
2*, 7	Uncached
Note: * These two values are required by the MIPS32 architecture. In the 4KE processor core, only values 0, 1, 2 and 3 are used. For example, values 4, 5 and 6 are not used and are mapped to 3. The value 7 is not used and is mapped to 2. Note that these values do have meaning in other MIPS Technologies processor implementations. Refer to the MIPS32 specification for more information.	

5.2.22 *Config1* Register (CP0 Register 16, Select 1)

The *Config1* register is an adjunct to the *Config* register and encodes additional information about capabilities present on the core. All fields in the *Config1* register are read-only.

The instruction and data cache configuration parameters include encodings for the number of sets per way, the line size, and the associativity. The total cache size for a cache is therefore:

Associativity * Line Size * Sets Per Way

If the line size is zero, there is no cache implemented.

Figure 5-24 *Config1* Register Format — Select 1

31	30 25	24 22	21 19	18 16	15 13	12 10	9 7	6	5	4	3	2	1	0
M	MMU Size	IS	IL	IA	DS	DL	DA	C2	MD	PC	WR	CA	EP	FP

Table 5-28 *Config1* Register Field Descriptions — Select 1

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
M	31	This bit is hardwired to ‘1’ to indicate the presence of the Config2 register.	R	1
MMU Size	30:25	This field contains the number of entries in the TLB minus one. The field is read as 15 decimal in the 4KEc core. The field is read as 0 decimal in the 4KEp and 4KEm cores, since no TLB is present.	R	Preset
IS	24:22	This field contains the number of instruction cache sets per way. Five options are available in the 4KE core. All others values are reserved: 0x0: 64 0x1: 128 0x2: 256 0x3: 512 0x4: 1024 0x5 - 0x7: Reserved	R	Preset
IL	21:19	This field contains the instruction cache line size. If an instruction cache is present, it must contain a fixed line size of 16 bytes. 0x0: No Icache present 0x3: 16 bytes 0x1, 0x2, 0x4 - 0x7: Reserved	R	Preset
IA	18:16	This field contains the level of instruction cache associativity. 0x0: Direct mapped 0x1: 2-way 0x2: 3-way 0x3: 4-way 0x4 - 0x7: Reserved	R	Preset

Table 5-28 *Config1* Register Field Descriptions — Select 1 (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
DS	15:13	This field contains the number of data cache sets per way. 0x0: 64 0x1: 128 0x2: 256 0x3: 512 0x4: 1024 0x5 - 0x7: Reserved	R	Preset
DL	12:10	This field contains the data cache line size. If a data cache is present, then it must contain a line size of 16 bytes. 0x0: No Dcache present 0x3: 16 bytes 0x1, 0x2, 0x4 - 0x7: Reserved	R	Preset
DA	9:7	This field contains the type of set associativity for the data cache. 0x0: Direct mapped 0x1: 2-way 0x2: 3-way 0x3: 4-way 0x4 - 0x7: Reserved	R	Preset
C2	6	Coprocessor 2 present. 0: No coprocessor is attached to the COP2 interface 1: A coprocessor is attached to the COP2 interface If the Cop2 interface logic is not implemented, this bit will read 0.	R	Preset
MD	5	MDMX implemented. This bit always reads as 0 because MDMX is not supported.	R	0
PC	4	Performance Counter registers implemented. Always a 0 since the 4KE core does not contain Performance Counters.	R	0
WR	3	Watch registers implemented. 0: No Watch registers are present 1: One or more Watch registers are present	R	Preset
CA	2	Code compression (MIPS16) implemented. 0: No MIPS16 present 1: MIPS16 is implemented	R	Preset
EP	1	EJTAG present: This bit is always set to indicate that the core implements EJTAG.	R	1
FP	0	FPU implemented. This bit is always zero since the core does not contain a floating point unit.	R	0

5.2.23 *Config2* Register (CP0 Register 16, Select 2)

The *Config2* register is an adjunct to the *Config* register and is reserved to encode additional capabilities information. *Config2* is allocated for showing the configuration of level 2/3 caches. These fields are reset to 0 because L2/L3 caches are not supported by the 4KE core. All fields in the *Config2* register are read-only.

Figure 5-25 *Config2* Register Format — Select 2

31	30 … 0
M	0

Table 5-29 *Config1* Register Field Descriptions — Select 1

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
M	31	This bit is hardwired to ‘1’ to indicate the presence of the Config3 register.	R	1
0	30:0	These bits are reserved.	R	0

5.2.24 *Config3* Register (CP0 Register 16, Select 3)

The *Config3* register encodes additional capabilities. All fields in the *Config3* register are read-only.

Figure 5-26 shows the format of the *Config3* register; Table 5-30 describes the *Config3* register fields.

Figure 5-26 Config3 Register Format

31	30 … 7	6	5	4	3 … 2	1	0
M	0 000 0000 0000 0000 0000 0000 0	VEIC	VInt	SP	0	SM	TL

Table 5-30 Config3 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>M</td><td>31</td><td>This bit is reserved to indicate that a Config4 register is present. With the current architectural definition, this bit should always read as a 0.</td><td>R</td><td>0</td></tr>
<tr><td>0</td><td>30:7,3:2</td><td>Must be written as zeros; returns zeros on read</td><td>0</td><td>0</td></tr>
<tr><td>VEIC</td><td>6</td><td>Support for an external interrupt controller is implemented.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Support for EIC interrupt mode is not implemented</td></tr><tr><td>1</td><td>Support for EIC interrupt mode is implemented</td></tr></table>The value of this bit is set by the static input, SI_EICPresent. This allows external logic to communicate whether an external interrupt controller is attached to the processor or not.</td><td>R</td><td>Externally Set</td></tr>
<tr><td>VInt</td><td>5</td><td>Vectored interrupts implemented. This bit indicates whether vectored interrupts are implemented.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Vector interrupts are not implemented</td></tr><tr><td>1</td><td>Vectored interrupts are implemented</td></tr></table>On the 4KE core, this bit is always a 1 since vectored interrupts are implemented.</td><td>R</td><td>1</td></tr>
<tr><td>SP</td><td>4</td><td>Small (1KByte) page support is implemented, and the PageGrain register exists. This bit will always read as 0 on the 4KEm and 4KEp cores, since no TLB is present.
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Small page support is not implemented</td></tr><tr><td>1</td><td>Small page support is implemented</td></tr></table></td><td>R</td><td>Preset</td></tr>
</table>

Table 5-30 Config3 Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
SM	1	SmartMIPS™ ASE implemented. This bit indicates whether the SmartMIPS ASE is implemented. Since SmartMIPS is not present on the 4KE core, this bit will always be 0. **Encoding** / **Meaning** 0: SmartMIPS ASE is not implemented 1: SmartMIPS ASE is implemented	R	0
TL	0	Trace Logic implemented. This bit indicates whether PC or data trace is implemented. **Encoding** / **Meaning** 0: Trace logic is not implemented 1: Trace logic is implemented	R	Preset

5.2.25 Load Linked Address (CP0 Register 17, Select 0)

The *LLAddr* register contains the physical address read by the most recent Load Linked (LL) instruction. This register is for diagnostic purposes only, and serves no function during normal operation.

Figure 5-27 *LLAddr* Register Format

31 28	27 0
0	PAddr[31:4]

Table 5-31 *LLAddr* Register Field Descriptions

Fields				
Name	**Bit(s)**	**Description**	**Read/ Write**	**Reset State**
0	31:28	Must be written as zeros; returns zeros on reads.	0	0
PAddr[31:4]	27:0	This field encodes the physical address read by the most recent Load Linked instruction.	R	Undefined

5.2.26 *WatchLo* Register (CP0 Register 18, Select 0)

The *WatchLo* and *WatchHi* registers together provide the interface to a watchpoint debug facility that initiates a watch exception if an instruction or data access matches the address specified in the registers. As such, they duplicate some functions of the EJTAG debug solution. Watch exceptions are taken only if the EXL and ERL bits are both zero in the *Status* register. If either bit is a one, the WP bit is set in the *Cause* register, and the watch exception is deferred until both the EXL and ERL bits are zero.

The *WatchLo* register specifies the base virtual address and the type of reference (instruction fetch, load, store) to match.

Figure 5-28 *WatchLo* Register Format

31 3	2	1	0
VAddr	I	R	W

Table 5-32 *WatchLo* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
VAddr	31:3	This field specifies the virtual address to match. Note that this is a doubleword address, since bits [2:0] are used to control the type of match.	R/W	Undefined
I	2	If this bit is set, watch exceptions are enabled for instruction fetches that match the address.	R/W	0 for Cold Reset only.
R	1	If this bit is set, watch exceptions are enabled for loads that match the address.	R/W	0 for Cold Reset only.
W	0	If this bit is set, watch exceptions are enabled for stores that match the address.	R/W	0 for Cold Reset only.

5.2.27 *WatchHi* Register (CP0 Register 19, Select 0)

The *WatchLo* and *WatchHi* registers together provide the interface to a watchpoint debug facility that initiates a watch exception if an instruction or data access matches the address specified in the registers. As such, they duplicate some functions of the EJTAG debug solution. Watch exceptions are taken only if the EXL and ERL bits are zero in the *Status* register. If either bit is a one, then the WP bit is set in the *Cause* register, and the watch exception is deferred until both the EXL and ERL bits are zero.

The *WatchHi* register contains information that qualifies the virtual address specified in the *WatchLo* register: an ASID, a Global (G) bit, and an optional address mask. If the G bit is 1, then any virtual address reference that matches the specified address will cause a watch exception. If the G bit is a 0, only those virtual address references for which the ASID value in the *WatchHi* register matches the ASID value in the *EntryHi* register cause a watch exception. The optional mask field provides address masking to qualify the address specified in *WatchLo*.

Figure 5-29 *WatchHi* Register Format

31	30	29 24	23 16	15 12	11 3	2 0
0	G	0	ASID	0	Mask	0

Table 5-33 *WatchHi* Register Field Descriptions

Fields				
Name	**Bit(s)**	**Description**	**Read/ Write**	**Reset State**
0	31	Must be written as zeros; returns zero on read.	0	0
G	30	If this bit is one, any address that matches that specified in the *WatchLo* register causes a watch exception. If this bit is zero, the ASID field of the *WatchHi* register must match the ASID field of the *EntryHi* register to cause a watch exception.	R/W	Undefined
0	29:24	Must be written as zeros; returns zeros on read.	0	0
ASID	23:16	ASID value which is required to match that in the *EntryHi* register if the G bit is zero in the *WatchHi* register.	R/W	Undefined
0	15:12	Must be written as zero; returns zero on read.	0	0
Mask	11:3	Bit mask that qualifies the address in the *WatchLo* register. Any bit in this field that is a set inhibits the corresponding address bit from participating in the address match.	R/W	Undefined
0	2:0	Must be written as zeros; returns zeros on read.	0	0

5.2.28 *Debug* Register (CP0 Register 23, Select 0)

The *Debug* register is used to control the debug exception and provide information about the cause of the debug exception and when re-entering at the debug exception vector due to a normal exception in debug mode. The read only information bits are updated every time the debug exception is taken or when a normal exception is taken when already in debug mode.

Only the DM bit and the EJTAGver field are valid when read from non-debug mode; the values of all other bits and fields are UNPREDICTABLE. Operation of the processor is UNDEFINED if the *Debug* register is written from non-debug mode.

Some of the bits and fields are only updated on debug exceptions and/or exceptions in debug mode, as shown below:

- DSS, DBp, DDBL, DDBS, DIB, DINT are updated on both debug exceptions and on exceptions in debug modes
- DExcCode is updated on exceptions in debug mode, and is undefined after a debug exception
- Halt and Doze are updated on a debug exception, and are undefined after an exception in debug mode
- DBD is updated on both debug and on exceptions in debug modes

All bits and fields are undefined when read from normal mode, except those explicitly described to be defined, e.g. EJTAGver and DM.

Figure 5-30 *Debug* Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19
DBD	DM	NoDCR	LSNM	Doze	Halt	CountDM	IBusEP	MCheckP	CacheEP	DBusEP	IEXI	DDBSImpr

18	17 15	14 10	9	8	7 6	5	4	3	2	1	0
DDBLImpr	Ver	DExcCode	NoSSt	SSt	R	DINT	DIB	DDBS	DDBL	DBp	DSS

Table 5-34 *Debug* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
DBD	31	Indicates whether the last debug exception or exception in debug mode, occurred in a branch delay slot: 0: Not in delay slot 1: In delay slot	R	Undefined
DM	30	Indicates that the processor is operating in debug mode: 0: Processor is operating in non-debug mode 1: Processor is operating in debug mode	R	0
NoDCR	29	Indicates whether the dseg memory segment is present: 0: dseg is present 1: No dseg present	R	0

Table 5-34 *Debug* Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
LSNM	28	Controls access of load/store between dseg and main memory: 0: Load/stores in dseg address range goes to dseg. 1: Load/stores in dseg address range goes to main memory.	R/W	0
Doze	27	Indicates that the processor was in any kind of low power mode when a debug exception occurred: 0: Processor not in low power mode when debug exception occurred 1: Processor in low power mode when debug exception occurred	R	Undefined
Halt	26	Indicates that the internal system bus clock was stopped when the debug exception occurred: 0: Internal system bus clock stopped 1: Internal system bus clock running	R	Undefined
CountDM	25	Indicates the Count register behavior in debug mode. 0: Count register stopped in debug mode 1: Count register is running in debug mode	R/W	1
IBusEP	24	Instruction fetch Bus Error exception Pending. Set when an instruction fetch bus error event occurs or if a 1 is written to the bit by software. Cleared when a Bus Error exception on instruction fetch is taken by the processor, and by reset. If IBusEP is set when IEXI is cleared, a Bus Error exception on instruction fetch is taken by the processor, and IBusEP is cleared.	R/W1	0
MCheckP	23	Indicates that an imprecise Machine Check exception is pending. All Machine Check exceptions are precise on the 4KE processors so this bit will always read as 0.	R	0
CacheEP	22	Indicates that an imprecise Cache Error is pending. Cache Errors cannot be taken by the 4KE cores so this bit will always read as 0	R	0
DBusEP	21	Data access Bus Error exception Pending. Covers imprecise bus errors on data access, similar to behavior of IBusEP for imprecise bus errors on an instruction fetch.	R/W1	0
IEXI	20	Imprecise Error eXception Inhibit controls exceptions taken due to imprecise error indications. Set when the processor takes a debug exception or exception in debug mode. Cleared by execution of the DERET instruction; otherwise modifiable by debug mode software. When IEXI is set, the imprecise error exception from a bus error on an instruction fetch or data access, cache error, or machine check is inhibited and deferred until the bit is cleared.	R/W	0
DDBSImpr	19	Indicates that an imprecise Debug Data Break Store exception was taken. All data breaks are precise on the 4KE cores, so this bit will always read as 0.	R	0
DDBLImpr	18	Indicates that an imprecise Debug Data Break Load exception was taken. All data breaks are precise on the 4KE cores, so this bit will always read as 0.	R	0

Table 5-34 *Debug* Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
Ver	17:15	EJTAG version.	R	010
DExcCode	14:10	Indicates the cause of the latest exception in debug mode. The field is encoded as the ExcCode field in the Cause register for those normal exceptions that may occur in debug mode. Value is undefined after a debug exception.	R	Undefined
NoSST	9	Indicates whether the single-step feature controllable by the SSt bit is available in this implementation: 0: Single-step feature available 1: No single-step feature available	R	0
SSt	8	Controls if debug single step exception is enabled: 0: No debug single-step exception enabled 1: Debug single step exception enabled	R/W	0
R	7:6	Reserved. Must be written as zeros; returns zeros on reads.	R	0
DINT	5	Indicates that a debug interrupt exception occurred. Cleared on exception in debug mode. 0: No debug interrupt exception 1: Debug interrupt exception	R/W	Undefined
DIB	4	Indicates that a debug instruction break exception occurred. Cleared on exception in debug mode. 0: No debug instruction exception 1: Debug instruction exception	R	Undefined
DDBS	3	Indicates that a debug data break exception occurred on a store. Cleared on exception in debug mode. 0: No debug data exception on a store 1: Debug instruction exception on a store	R	Undefined
DDBL	2	Indicates that a debug data break exception occurred on a load. Cleared on exception in debug mode. 0: No debug data exception on a load 1: Debug instruction exception on a load	R	Undefined
DBp	1	Indicates that a debug software breakpoint exception occurred. Cleared on exception in debug mode. 0: No debug software breakpoint exception 1: Debug software breakpoint exception	R	Undefined
DSS	0	Indicates that a debug single-step exception occurred. Cleared on exception in debug mode. 0: No debug single-step exception 1: Debug single-step exception	R	Undefined

5.2.29 *Trace Control* Register (CP0 Register 23, Select 1)

The *TraceControl* register configuration is shown below. Note the special behavior of the ASID_M, ASID, and G fields for the 4KEm and 4KEp processors.

This register is only implemented if the EJTAG Trace capability is present.

Figure 5-31 *Trace Control* Register Format

31	30	29 28	27	26	25	24	23	22	21	20 13	12 5	4	3 1	0
TS	UT	0	TB	IO	D	E	K	S	U	ASID_M	ASID	G	Mode	On

Table 5-35 *TraceControl* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
TS	31	The trace select bit is used to select between the hardware and the software trace control bits. A value of zero selects the external hardware trace block signals, and a value of one selects the trace control bits in this software control register.	R/W	0
UT	30	This bit is used to indicate the type of user-triggered trace record. A value of zero implies a user type 1 and a value of one implies a user type 2. The actual triggering of a user trace record happens on a write to the *UserTraceData* register.	R/W	Undefined
0	29:28	Reserved for future use; Must be written as zero; returns zero on read.	0	0
TB	27	Trace All Branch. When set to one, this tells the processor to trace the PC value for all taken branches, not just the ones whose branch target address is statically unpredictable.	R/W	Undefined
IO	26	Inhibit Overflow. This signal is used to indicate to the core trace logic that slow but complete tracing is desired. When set to one, the core tracing logic does not allow a FIFO overflow or discard trace data. This is achieved by stalling the pipeline when the FIFO is nearly full, so that no trace records are ever lost.	R/W	Undefined
D	25	When set to one, this enables tracing in Debug Mode (see Section 9.7.1, "Processor Modes" on page 205). For trace to be enabled in Debug mode, the On bit must be one, and either the G bit must be one, or the current process ASID must match the ASID field in this register. When set to zero, trace is disabled in Debug Mode, irrespective of other bits.	R/W	Undefined

Table 5-35 *TraceControl* Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
E	24	When set to one, this enables tracing in Exception Mode (see Section 9.7.1, "Processor Modes" on page 205). For trace to be enabled in Exception mode, the On bit must be one, and either the G bit must be one, or the current process ASID must match the ASID field in this register. When set to zero, trace is disabled in Exception Mode, irrespective of other bits.	R/W	Undefined
K	23	When set to one, this enables tracing in Kernel Mode (see Section 9.7.1, "Processor Modes" on page 205). For trace to be enabled in Kernel mode, the On bit must be one, and either the G bit must be one, or the current process ASID must match the ASID field in this register. When set to zero, trace is disabled in Kernel Mode, irrespective of other bits.	R/W	Undefined
0	22	This bit is reserved. Must be written as zero; returns zero on read.	0	0
U	21	When set to one, this enables tracing in User Mode (see Section 9.7.1, "Processor Modes" on page 205). For trace to be enabled in User mode, the On bit must be one, and either the G bit must be one, or the current process ASID must match the ASID field in this register. When set to zero, trace is disabled in User Mode, irrespective of other bits.	R/W	Undefined
ASID_M	20:13	This is a mask value applied to the ASID comparison (done when the G bit is zero). A “1” in any bit in this field inhibits the corresponding ASID bit from participating in the match. As such, a value of zero in this field compares all bits of ASID. Note that the ability to mask the ASID value is not available in the hardware signal bit; it is only available via the software control register. In the 4KEm and 4KEp cores where ASID is not supported, this field is ignored on write and returns zero on read.	R/W	Undefined
ASID	12:5	The ASID field to match when the G bit is zero. When the G bit is one, this field is ignored. In the 4KEm and 4KEp cores where ASID is not supported, this field is ignored on write and returns zero on read.	R/W	Undefined
G	4	Global bit. When set to one, tracing is to be enabled for all processes, provided that other enabling functions (like U, S, etc.,) are also true. In the 4KEm and 4KEp cores where ASID is not supported, this field is ignored on write and returns 1 on read. This causes all match equations to work correctly in the absence of an ASID.	R/W	Undefined

Table 5-35 *TraceControl* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Mode</td><td>3:1</td><td>These three bits control the trace mode function.
<table>
<tr><th>Mode</th><th>Trace Mode</th></tr>
<tr><td>000</td><td>Trace PC</td></tr>
<tr><td>001</td><td>Trace PC and load address</td></tr>
<tr><td>010</td><td>Trace PC and store address</td></tr>
<tr><td>011</td><td>Trace PC and both load/store addresses</td></tr>
<tr><td>100</td><td>Trace PC and load data</td></tr>
<tr><td>101</td><td>Trace PC and load address and data</td></tr>
<tr><td>110</td><td>Trace PC and store address and data</td></tr>
<tr><td>111</td><td>Trace PC and both load/store address and data</td></tr>
</table>
The $\textit{TraceControl2}_{\text{ValidModes}}$ field determines which of these encodings are supported by the processor. The operation of the processor is UNPREDICTABLE if this field is set to a value which is not supported by the processor.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>On</td><td>0</td><td>This is the master trace enable switch in software control. When zero, tracing is always disabled. When set to one, tracing is enabled whenever the other enabling functions are also true.</td><td>R/W</td><td>0</td></tr>
</table>

5.2.30 *Trace Control2* Register (CP0 Register 23, Select 2)

The *TraceControl2* register provides additional control and status information. Note that some fields in the *TraceControl2* register are read-only, but have a reset state of “Undefined”. This is because these values are loaded from the Trace Control Block (TCB) (see Section 9.9, "Trace Control Block (TCB) Registers (hardware control)" on page 209). As such, these fields in the *TraceControl2* register will not have valid values until the TCB asserts these values.

This register is only implemented if the EJTAG Trace capability is present.

Figure 5-32 *Trace Control2* Register Format

31 … 7	6 … 5	4	3	2 … 0
0	Valid Modes	TBI	TBU	SyP

Table 5-36 *TraceControl2* Register Field Descriptions

Fields: Name	Fields: Bits	Description	Read/ Write	Reset State
0	31:5	Reserved for future use; Must be written as zero; returns zero on read.	0	0
ValidModes	6:5	This field specifies the type of tracing that is supported by the processor, as follows: **Encoding** / **Meaning** 00: PC tracing only 01: PC and load and store address tracing only 10: PC, load and store address, and load and store data 11: Reserved	R	10
TBI	4	This bit indicates how many trace buffers are implemented by the TCB, as follows: **Encoding** / **Meaning** 0: Only one trace buffer is implemented, and the TBU bit of this register indicates which trace buffer is implemented 1: Both on-chip and off-chip trace buffers are implemented by the TCB and the TBU bit of this register indicates to which trace buffer the trace is currently written.	R	Per implementation

Table 5-36 *TraceControl2* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>TBU</td><td>3</td><td>This bit denotes to which trace buffer the trace is currently being written and is used to select the appropriate interpretation of the TraceControl2$_{SyP}$ field.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Trace data is being sent to an on-chip trace buffer</td></tr>
<tr><td>1</td><td>Trace Data is being sent to an off-chip trace buffer</td></tr>
</table></td><td>R</td><td>Undefined</td></tr>
<tr><td>SyP</td><td>2:0</td><td>Used to indicate the synchronization period.

The period (in cycles) between which the periodic synchronization information is to be sent is defined as shown below, for both when the trace buffer is on-chip and off-chip.
<table>
<tr><th>SyP</th><th>On-chip</th><th>Off-chip</th></tr>
<tr><td>000</td><td>2^2</td><td>2^7</td></tr>
<tr><td>001</td><td>2^3</td><td>2^8</td></tr>
<tr><td>010</td><td>2^4</td><td>2^9</td></tr>
<tr><td>011</td><td>2^5</td><td>2^{10}</td></tr>
<tr><td>100</td><td>2^6</td><td>2^{11}</td></tr>
<tr><td>101</td><td>2^7</td><td>2^{12}</td></tr>
<tr><td>110</td><td>2^8</td><td>2^{13}</td></tr>
<tr><td>111</td><td>2^9</td><td>2^{14}</td></tr>
</table>
The “On-chip” column value is used when the trace data is being written to an on-chip trace buffer (e.g, TraceControl2$_{TBU}$ = 0). Conversely, the “Off-chip” column is used when the trace data is being written to an off-chip trace buffer (e.g, TraceControl2$_{TBU}$ = 1).</td><td>R</td><td>Undefined</td></tr>
</table>

5.2.31 *User Trace Data* Register (CP0 Register 23, Select 3)

A software write to any bits in the *UserTraceData* register will trigger a trace record to be written indicating a type 1 or type 2 user format. The type is based on the UT bit in the *TraceControl* register. This register cannot be written in consecutive cycles. The trace output data is UNPREDICTABLE if this register is written in consecutive cycles.

This register is only implemented if the EJTAG Trace capability is present.

Figure 5-33 *User Trace Data* Register Format

31 ... 0
Data

Table 5-37 *UserTraceData* Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read/ Write**	**Reset State**
Data	31:0	Software readable/writable data. When written, this triggers a user format trace record out of the PDtrace interface that transmits the Data field to trace memory.	R/W	0

5.2.32 *TraceBPC* Register (CP0 Register 23, Select 4)

This register is used to control start and stop of tracing using an EJTAG Hardware breakpoint. The Hardware breakpoint would then be set as a trigger source and optionally also as a Debug exception breakpoint.

This register is only implemented if both Hardware breakpoints and the EJTAG Trace capability are present.

Figure 5-34 *Trace BPC* Register Format

31	30 18	17 16	15	14 4	3 0
DE	0	DBPOn	IE	0	IBPOn

Table 5-38 *TraceBPC* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
DE	31	Used to specify whether the trigger signal from EJTAG data breakpoint should trigger tracing functions or not: 0: disables trigger signals from data breakpoints 1: enables trigger signals from data breakpoints	R/W	0
0	30:18	Reserved	0	0
DBPOn	30:16	Each of the 2 bits corresponds to the 2 possible EJTAG hardware data breakpoints that may be implemented. For example, bit 16 corresponds to the first data breakpoint. If 2 data breakpoints are present in the EJTAG implementation, then they correspond to bits 16 and 17. The rest are always ignored by the tracing logic since they will never be triggered. A value of one for each bit implies that a trigger from the corresponding data breakpoint should start tracing. And a value of zero implies that tracing should be turned off with the trigger signal.	R/W	0
IE	15	Used to specify whether the trigger signal from EJTAG instruction breakpoint should trigger tracing functions or not: 0: disables trigger signals from instruction breakpoints 1: enables trigger signals from instruction breakpoints	R/W	0
0	14:4	Reserved	0	0
IBPOn	3:0	Each of the 4 bits corresponds to the 4 possible EJTAG hardware instruction breakpoints that may be implemented. Bit 0 corresponds to the first instruction breakpoint, and so on. If only 2 instruction breakpoints are present in the EJTAG implementation, then only bits 0 and 1 are used. The rest are always ignored by the tracing logic since they will never be triggered. A value of one for each bit implies that a trigger from the corresponding instruction breakpoint should start tracing. And a value of zero implies that tracing should be turned off with the trigger signal.	R/W	0

5.2.33 Debug Exception Program Counter Register (CP0 Register 24, Select 0)

The Debug Exception Program Counter (*DEPC*) register is a read/write register that contains the address at which processing resumes after a debug exception or debug mode exception has been serviced.

For synchronous (precise) debug and debug mode exceptions, the *DEPC* contains either:

- The virtual address of the instruction that was the direct cause of the debug exception, or
- The virtual address of the immediately preceding branch or jump instruction, when the debug exception causing instruction is in a branch delay slot, and the Debug Branch Delay (DBD) bit in the *Debug* register is set.

For asynchronous debug exceptions (debug interrupt), the *DEPC* contains the virtual address of the instruction where execution should resume after the debug handler code is executed.

In processors that implement the MIPS16 ASE, a read of the DEPC register (via MFC0) returns the following value in the destination GPR:

$$\texttt{GPR[rt]} \leftarrow \texttt{DebugExceptionPC}_{31..1} \;||\; \texttt{ISAMode}_0$$

That is, the upper 31 bits of the debug exception PC are combined with the lower bit of the ISAMode field and written to the GPR.

Similarly, a write to the DEPC register (via MTC0) takes the value from the GPR and distributes that value to the debug exception PC and the ISAMode field, as follows

$$\texttt{DebugExceptionPC} \leftarrow \texttt{GPR[rt]}_{31..1} \;||\; 0$$
$$\texttt{ISAMode} \leftarrow \texttt{2\#0} \;||\; \texttt{GPR[rt]}_0$$

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the debug exception PC, and the lower bit of the debug exception PC is cleared. The upper bit of the ISAMode field is cleared and the lower bit is loaded from the lower bit of the GPR.

Figure 5-35 *DEPC* Register Format

31 … 0
DEPC

Table 5-39 *DEPC* Register Formats

Fields		Description	Read/ Write	Reset
Name	Bit(s)			
DEPC	31:0	The *DEPC* register is updated with the virtual address of the instruction that caused the debug exception. If the instruction is in the branch delay slot, then the virtual address of the immediately preceding branch or jump instruction is placed in this register. Execution of the DERET instruction causes a jump to the address in the *DEPC*.	R/W	Undefined

5.2.34 *ErrCtl* Register (CP0 Register 26, Select 0)

The ErrCtl register provides a mechanism for enabling software testing of the way-select and data RAM arrays for both the ICache and DCache. The way-selection RAM test mode is enabled by setting the WST bit. It modifies the functionality of the CACHE Index Load Tag and Index Store Tag operations so that they modify the way-selection RAM and leave the Tag RAMs untouched. When this bit is set, the lower 6 bits of the PA field in the TagLo register are used as the source and destination for Index Load Tag and Index Store Tag CACHE operations.

The WST bit also enables the data RAM test mode. When this bit is set, the Index Store Data CACHE instruction is enabled. This CACHE operation writes the contents of the DataLo register to the word in the data array that is indicated by the index and byte address.

The SPR bit enables CACHE accesses to the optional Scratchpad RAMs. When this bit is set, Index Load Tag, Index Store Tag, and Index Store Data CACHE instructions will send reads or writes to the Scratchpad RAM port. The effects of these operations are dependent on the particular Scratchpad implementation.

Figure 5-36 *ErrCtl* Register Format

31 30	29	28	27 0
R	WST	SPR	R

Table 5-40 *ErrCtl* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
WST	29	Indicates whether the tag array or the way-select array should be read/written on Index Load/Store Tag CACHE instructions. Also enables the Index Store Data CACHE instruction which writes the contents of DataLo to the data array.	R/W	0
SPR	28	Forces indexed CACHE instructions to operate on the ScratchPad RAM instead of the cache	R/W	0
R	31:30, 27:0	Must be written as zero; returns zero on reads.	0	0

5.2.35 *TagLo* Register (CP0 Register 28, Select 0)

The *TagLo* register acts as the interface to the cache tag array. The Index Store Tag and Index Load Tag operations of the CACHE instruction use the *TagLo* register as the source of tag information. Note that the 4KE core does not implement the TagHi register.

Figure 5-37 *TagLo* Register Format

31 16	15 10	9 8	7	6	5	4 0
PA	PA/LRU	R	V	D	L	R

Table 5-41 *TagLo* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
PA	31:10	This field contains the physical address of the cache line. Bit 31 corresponds to bit 31 of the PA and bit 10 corresponds to bit 10 of the PA.	R/W	Undefined
LRU	15:10	This field contains the value read from or to be stored to the WS array if the WST bit in the ErrCtl register is set.	R/W	Undefined
R	9:8, 4:0	Must be written as zero; returns zero on read.	0	0
V	7	This field indicates whether the cache line is valid.	R/W	Undefined
D	6	This field indicates whether the cache line is dirty. It will only be set if bit 7 (valid) is also set.	R/W	Undefined
L	5	Specifies the lock bit for the cache tag. When this bit is set, and the valid bit is set, the corresponding cache line will not be replaced by the cache replacement algorithm.	R/W	Undefined

5.2.36 *DataLo* Register (CP0 Register 28, Select 1)

The *DataLo* register is a register that acts as the interface to the cache data array and is intended for diagnostic operations only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *DataLo* register. If the WST bit in the *ErrCtl* register is set, then the contents of *DataLo* can be written to the cache data array by doing an Index Store Data CACHE instruction. Note that the 4KE core does not implement the DataHi register.

Figure 5-38 *DataLo* Register Format

31	0
DATA	

Table 5-42 *DataLo* Register Field Description

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
DATA	31:0	Low-order data read from the cache data array.	R/W	Undefined

5.2.37 *ErrorEPC* (CP0 Register 30, Select 0)

The *ErrorEPC* register is a read/write register, similar to the *EPC* register, except that *ErrorEPC* is used on error exceptions. All bits of the *ErrorEPC* register are significant and must be writable. It is also used to store the program counter on Reset, Soft Reset, and nonmaskable interrupt (NMI) exceptions.

The *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. This address can be:

- The virtual address of the instruction that caused the exception
- The virtual address of the immediately preceding branch or jump instruction when the error causing instruction is in a branch delay slot

Unlike the *EPC* register, there is no corresponding branch delay slot indication for the *ErrorEPC* register.

In processors that implement the MIPS16 ASE, a read of the ErrorEPC register (via MFC0) returns the following value in the destination GPR:

$$\texttt{GPR[rt]} \leftarrow \texttt{ErrorExceptionPC}_{31..1} \;||\; \texttt{ISAMode}_0$$

That is, the upper 31 bits of the error exception PC are combined with the lower bit of the ISAMode field and written to the GPR.

Similarly, a write to the ErrorEPC register (via MTC0) takes the value from the GPR and distributes that value to the error exception PC and the ISAMode field, as follows

$$\texttt{ErrprExceptionPC} \leftarrow \texttt{GPR[rt]}_{31..1} \;||\; \texttt{0}$$
$$\texttt{ISAMode} \leftarrow \texttt{2\#0} \;||\; \texttt{GPR[rt]}_0$$

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the error exception PC, and the lower bit of the error exception PC is cleared. The upper bit of the ISAMode field is cleared and the lower bit is loaded from the lower bit of the GPR.

Figure 5-39 *ErrorEPC* Register Format

31 ... 0
ErrorEPC

Table 5-43 *ErrorEPC* Register Field Description

Fields				
Name	Bit(s)	Description	Read/ Write	Reset State
ErrorEPC	31:0	Error Exception Program Counter.	R/W	Undefined

5.2.38 *DeSave* Register (CP0 Register 31, Select 0)

The Debug Exception Save (*DeSave*) register is a read/write register that functions as a simple memory location. This register is used by the debug exception handler to save one of the GPRs that is then used to save the rest of the context to a pre-determined memory area (such as in the EJTAG Probe). This register allows the safe debugging of exception handlers and other types of code where the existence of a valid stack for context saving cannot be assumed.

Figure 5-40 *DeSave* Register Format

31   0
DESAVE

Table 5-44 *DeSave* Register Field Description

Fields				
Name	**Bit(s)**	**Description**	**Read/ Write**	**Reset State**
DESAVE	31:0	Debug exception save contents.	R/W	Undefined

Hardware and Software Initialization

A MIPS32™ 4KE™ processor core contains only a minimal amount of hardware initialization and relies on software to fully initialize the device.

This chapter contains the following sections:

- Section 6.1, "Hardware-Initialized Processor State"
- Section 6.2, "Software Initialized Processor State"

6.1 Hardware-Initialized Processor State

A 4KE processor core, like most other MIPS processors, is not fully initialized by hardware reset. Only a minimal subset of the processor state is cleared. This is enough to bring the core up while running in unmapped and uncached code space. All other processor state can then be initialized by software. *SI_ColdReset* is asserted after power-up to bring the device into a known state. Soft reset can be forced by asserting the *SI_Reset* pin. This distinction is made for compatibility with other MIPS processors. In practice, both resets are handled identically with the exception of the setting of $Status_{SR}$.

6.1.1 Coprocessor 0 State

Much of the hardware initialization occurs in Coprocessor 0.

- *Random* (4KEc core only)- cleared to maximum value on Reset/SoftReset
- *Wired* (4KEc core only)- cleared to 0 on Reset/SoftReset
- $Status_{BEV}$ - cleared to 1 on Reset/SoftReset
- $Status_{TS}$ - cleared to 0 on Reset/SoftReset
- $Status_{SR}$ - cleared to 0 on Reset, set to 1 on SoftReset
- $Status_{NMI}$ - cleared to 0 on Reset/SoftReset
- $Status_{ERL}$ - set to 1 on Reset/SoftReset
- $Status_{RP}$ - cleared to 0 on Reset/SoftReset
- $WatchLo_{I,R,W}$ - cleared to 0 on Reset/SoftReset
- *Config* fields related to static inputs - set to input value by Reset/SoftReset
- $Config_{K0}$ - set to 010 (uncached) on Reset/SoftReset
- $Config_{KU}$ - set to 010 (uncached) on Reset/SoftReset (4KEm™ and 4KEp™ cores only)
- $Config_{K23}$ - set to 010 (uncached) on Reset/SoftReset (4KEm and 4KEp cores only)
- ContextConfig - set to 0x007ffff0 on Reset/SoftReset (MIPS32 configuration)
- $\text{PageGrain}_{\text{Mask}}$ - set to 11 on Reset/SoftReset (MIPS32 compatibility mode)
- *DebugDM* - cleared to 0 on Reset/SoftReset (unless EJTAGBOOT option is used to boot into DebugMode, see Chapter 9, "EJTAG Debug Support." for details)

- $Debug_{LSNM}$ - cleared to 0 on Reset/SoftReset
- $Debug_{IBusEP}$ - cleared to 0 on Reset/SoftReset
- $Debug_{DBusEP}$ - cleared to 0 on Reset/SoftReset
- $Debug_{IEXI}$ - cleared to 0 on Reset/SoftReset
- $Debug_{SSt}$ - cleared to 0 on Reset/SoftReset

6.1.2 TLB Initialization (4KEc core only)

Each 4KEc TLB entry has a “hidden” state bit which is set by Reset/SoftReset and is cleared when the TLB entry is written. This bit disables matches and prevents “TLB Shutdown” conditions from being generated by the power-up values in the TLB array (when two or more TLB entries match on a single address). This bit is not visible to software.

6.1.3 Bus State Machines

All pending bus transactions are aborted and the state machines in the bus interface unit are reset when a Reset or SoftReset exception is taken.

6.1.4 Static Configuration Inputs

All static configuration inputs (defining the bus mode and cache size for example) should only be changed during Reset.

6.1.5 Fetch Address

Upon Reset/SoftReset, unless the EJTAGBOOT option is used, the fetch is directed to VA 0xBFC00000 (PA 0x1FC00000). This address is in KSeg1,which is unmapped and uncached, so that the TLB and caches do not require hardware initialization.

6.2 Software Initialized Processor State

Software is required to initialize the following parts of the device.

6.2.1 Register File

The register file powers up in an unknown state with the exception of r0 which is always 0. Initializing the rest of the register file is not required for proper operation. Good code will generally not read a register before writing to it, but the boot code can initialize the register file for added safety.

6.2.2 TLB (4KEc Core Only)

Because of the hidden bit indicating initialization, the 4KEc core does not require TLB initialization upon ColdReset. This is an implementation specific feature of the 4KEc core and cannot be relied upon if writing generic code for MIPS32/64 processors. When initializing the TLB, care must be taken to avoid creating a “TLB Shutdown” condition where two TLB entries could match on a single address. Unique virtual addresses should be written to each TLB entry to avoid this.

6.2.3 Caches

The cache tag and data arrays power up to an unknown state and are not affected by reset. Every tag in the cache arrays should be initialized to an invalid state using the CACHE instruction (typically the Index Invalidate function). This can be a long process, especially since the instruction cache initialization needs to be run in an uncached address region.

6.2.4 Coprocessor 0 State

Miscellaneous COP0 states need to be initialized prior to leaving the boot code. There are various exceptions which are blocked by ERL=1 or EXL=1 and which are not cleared by Reset. These can be cleared to avoid taking spurious exceptions when leaving the boot code.

- *Cause*: WP (Watch Pending), SW0/1 (Software Interrupts) should be cleared.
- *Config*: K0 should be set to the desired Cache Coherency Algorithm (CCA) prior to accessing Kseg0.
- *Config*: (4KEm and 4KEp cores only) KU and K23 should be set to the desired CCA for USeg/KUSeg and KSeg2/3 respectively prior to accessing those regions.
- *Count*: Should be set to a known value if Timer Interrupts are used.
- *Compare*: Should be set to a known value if Timer Interrupts are used. The write to compare will also clear any pending Timer Interrupts (Thus, *Count* should be set before *Compare* to avoid any unexpected interrupts).
- *Status*: Desired state of the device should be set.
- Other COP0 state: Other registers should be written before they are read. Some registers are not explicitly writeable, and are only updated as a by-product of instruction execution or a taken exception. Uninitialized bits should be masked off after reading these registers.

Caches

This chapter describes the caches present in a MIPS32 4KE processor core. It contains the following sections:

- Section 7.1, "Cache Configurations"
- Section 7.2, "Cache Protocols"
- Section 7.3, "Instruction Cache"
- Section 7.4, "Data Cache"
- Section 7.5, "CACHE Instruction"
- Section 7.6, "Software Cache Testing"
- Section 7.7, "Memory Coherence Issues"

7.1 Cache Configurations

A 4KE processor core supports separate instruction and data caches which may be flexibly configured at build time for various sizes, organizations and set-associativities. The use of separate caches allows instruction and data references to proceed simultaneously. Both caches are virtually indexed and physically tagged, allowing cache access to occur in parallel with virtual-to-physical address translation.

The instruction and data caches are independently configured. For example, the data cache can be 2 KB in size and 2-way set associative, while the instruction cache can be 8 KB in size and 4-way set associative. Each cache is accessed in a single processor cycle.

Cache refills are performed using a 4-word fill buffer, which holds data returned from memory during a 4-beat burst transaction. The critical miss word is always returned first. The caches are blocking until the critical word is returned, but the pipeline may proceed while the other 3 beats of the burst are still active on the bus.

Table 7-1 lists the instruction and data cache attributes:

Table 7-1 Instruction and Data Cache Attributes

Parameter	Instruction	Data
Size	0 - 64 KB	0 - 64 KB
Number of Cache Sets	0, 64, 128, 256, 512 and 1024	0, 64, 128, 256, 512 and 1024
Lines Per Set (Associativity)	1 - 4 way set associative	1 - 4 way set associative
Line Size	16 Bytes	16 Bytes
Read Unit	32 bits	32 bits
Minimum Write Unit	32 bits	8 bits

Table 7-1 Instruction and Data Cache Attributes (Continued)

Parameter	Instruction	Data
Write Policy	N/A	Software selectable options: • write-back with write-allocate • write-through with write-allocate • write-through without write-allocate
Miss restart after transfer of	miss word	miss word
Cache Locking	per line	per line

Table 7-2 shows the cache size and organization options; note that the same total cache size may be achieved with several different set associativities. Software can identify the instruction or data cache configuration on a 4KE core by reading the appropriate bits of the *Config1* register; see Section 5.2.22, "Config1 Register (CP0 Register 16, Select 1)" on page 125.

Table 7-2 Instruction and Data Cache Sizes

Cache Size (bytes)	Way Organization Options
0K	No cache
1K	One 1K way
2K	One 2K way Two 1K ways
3K	Three 1K ways
4K	One 4K way Two 2K ways Four 1K ways
6K	Three 2K ways
8K	One 8K way Two 4K ways Four 2K ways
12K	Three 4K ways
16K	One 16K way Two 8K ways Four 4K ways
24K	Three 8K ways
32K	Two 16K ways Four 8K ways
48K	Three 16K ways
64K	Four 16K ways

7.2 Cache Protocols

This section describes cache organization, attributes, and cache-line replacement for the instruction and data caches. This section also discusses issues relating to virtual aliasing.

7.2.1 Cache Organization

The instruction and data caches each consist of three arrays: tag, data and way-select. The caches are virtually indexed, since a virtual address is used to select the appropriate line within each of the three arrays. The caches are physically tagged, as the tag array contains a physical, not virtual, address.

The tag and data arrays hold *n* ways of information per set, corresponding to the *n*-way set associativity of the cache, where *n* can be between 1 and 4 for a cache in a 4KE core. The way-select array holds information to choose the way to be filled, as well as dirty bits in the case of the data cache.

Figure 7-1 on page 155 shows the format of each line in the tag, data and way-select arrays.

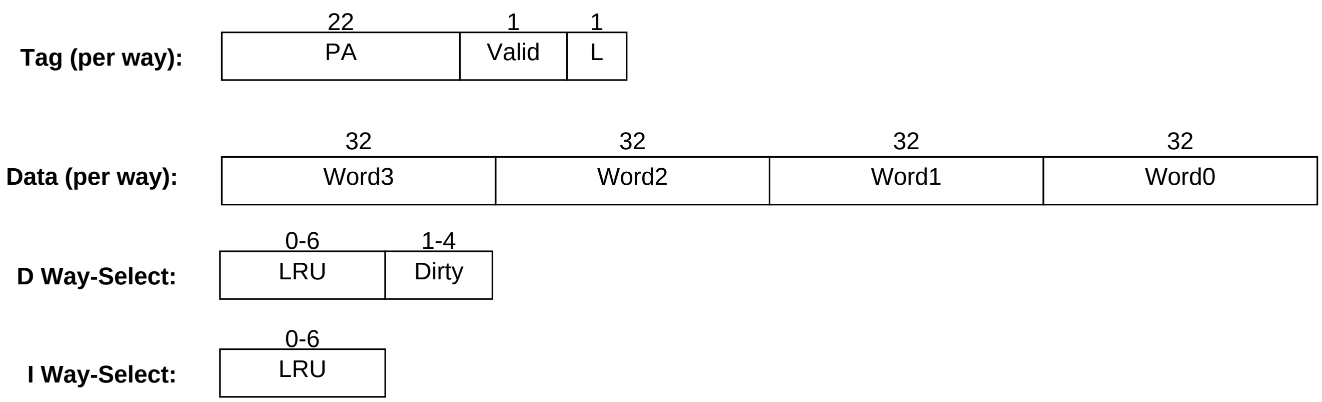


Figure 7-1 Cache Array Formats

A tag entry consists of the upper 22 bits of the physical address (bits [31:10]), one valid bit for the line, and a lock bit. A data entry contains the four 32-bit words in the line, for a total of 16 bytes. All four words in the line are present or not in the data array together, hence the single valid bit stored with the tag. Once a valid line is resident in the cache, byte, halfword, triple-byte or full word stores can update all or a portion of the words in that line. The tag and data entries are repeated for each of the *n* lines in the set, per the associativity.

A way-select entry holds bits choosing the way to be replaced according to a Least Recently Used (LRU) algorithm. The LRU information applies to all the ways and there is one way-select entry for all the ways in the set. The number of bits in the way-select entry depends on the set associativity. In a direct mapped cache (*n*=1), there is no need for LRU bits, since fills can only go to one place only. Table 7-3 shows the number of LRU bits required as a function of associativity. The array with way-select entries for the data cache also holds dirty bit(s) for the lines. One dirty bit is required per line, as shown in Table 7-3. The instruction cache only supports reads, hence only LRU entries are stored in the instruction way-select array.

Table 7-3 LRU and Dirty Width in Way-Select Array

Associativity (*n*)	LRU Bits	Dirty Bits (data cache only)
1	0	1
2	1	2

Table 7-3 LRU and Dirty Width in Way-Select Array

Associativity (*n*)	LRU Bits	Dirty Bits (data cache only)
3	3	3
4	6	4

7.2.2 Cacheability Attributes

A 4KE core supports the following cacheability attributes:

- *Uncached*: Addresses in a memory area indicated as uncached are not read from the cache. Stores to such addresses are written directly to main memory, without changing cache contents.
- *Write-back with write allocation*: Loads and instruction fetches first search the cache, reading main memory only if the desired data does not reside in the cache. On data store operations, the cache is first searched to see if the target address is cache resident. If it is resident, the cache contents are updated, but main memory is not written. If the cache lookup misses on a store, main memory is read to bring the line into the cache and merge it with the new store data. Hence, the allocation policy on a cache miss is read- or write-allocate. Data stores will update the appropriate dirty bit in the way-select array to indicate that the line contains modified data. When a line with dirty data is displaced from the cache, it is written back to memory.
- *Write-through with no write allocation*: Loads and instruction fetches first search the cache, reading main memory only if the desired data does not reside in the cache. On data store operations, the cache is first searched to see if the target address is cache resident. If it is resident, the cache contents are updated, and main memory is also written. If the cache lookup misses on a store, only main memory is written. Hence, the allocation policy on a cache miss is read-allocate only.
- *Write-through with write allocation*: Loads and instruction fetches first search the cache, reading main memory only if the desired data does not reside in the cache. On data store operations, the cache is first searched to see if the target address is cache resident. If it is resident, the cache contents are updated, and main memory is also written. If the cache lookup misses on a store, main memory is read to bring the line into the cache and merge it with the new store data. In addition, the store data is also written to main memory. Hence, the allocation policy on a cache miss is read- or write-allocate.

Some segments of memory employ a fixed caching policy; for example the kseg1 is always uncacheable. Other segments of memory allow the caching policy to be selected by software. Generally, the cache policy for these programmable regions is defined by a cacheability attribute field associated with that region of memory. See Chapter 3, “Memory Management,” on page 33 for further details.

7.2.3 Replacement Policy

The replacement policy refers to how a way is chosen to hold an incoming cache line on a miss which will result in a cache fill, when a cache is at least two-way set associative. In a direct mapped cache (one-way set associative), the replacement policy is irrelevant since there is only one way available. The replacement policy is least recently used (LRU), but excluding any locked ways. The LRU bit(s) in the way-select array encode the order in which ways on that line have been accessed.

On a cache miss, the lock and LRU bits for the tag and way-select entries of the selected line may be used to determine the way which will be chosen. The number of lock bits and the number of LRU bits depend on the set associativity of the cache.

The LRU field in the way select array is updated as follows:

- On a cache hit, the associated way is updated to be the most recently used. The order of the other ways relative to each another is unchanged.
- On a cache refill, the filled way is updated to be the most recently used.
- On CACHE instructions, the update of the LRU bits depends on the type of operation to be performed:
  - **Index (Writeback) Invalidate**: Least-recently used.
  - **Index Load Tag**: No update.
  - **Index Store Tag, WST=0:** Most-recently used if valid bit is set in *TagLo* CP0 register. Least-recently used if valid bit is cleared in *TagLo* CP0 register.
  - **Index Store Tag, WST=1:** Update the field with the contents of the *TagLo* CP0 register (refer to Table 7-5, Table 7-6 or Table 7-7 for the valid values of this field).
  - **Index Store Data:** No update.
  - **Hit Invalidate:** Least-recently used if a hit is generated, otherwise unchanged.
  - **Fill:** Most-recently used.
  - **Hit (Writeback) Invalidate:** Least-recently used if a hit is generated, otherwise unchanged.
  - **Hit Writeback:** No update.
  - **Fetch and Lock:** Most-recently used.

If all ways are valid, then any locked ways will be excluded from consideration for replacement. For the unlocked ways, the LRU bits are used to identify the way which has been used least recently, and that way is selected for replacement. If all ways are locked: Fill data will not fill into the cache, and Write-back stores turn into Write-through Write-allocate stores.

If the way selected for replacement has its dirty bit asserted in the way-select array, then that 16-byte line will be written back to memory before the new fill can occur.

7.2.4 Virtual Aliasing

Since the caches are virtually indexed and physically tagged, a potential issue referred to as *virtual aliasing* might exist. Virtual aliasing occurs if the virtual bits used to index a cache array are not consistent with the overlapping physical bits, after the virtual address has been translated to a physical address. The possibility of virtual aliasing only occurs in address regions which are mapped through a TLB-based memory management unit, so it is only relevant for the 4KEc core and cannot occur in the 4KEm or 4KEp cores which contain a fixed memory management unit.

In TLB-mapped address regions, virtual aliasing may occur if the cache size per way is greater than the page size. For example, consider a 16 KB cache organized as 2-way set associative. The size per way is then 8 KB, so virtual address bits [12:0] are used to index the array. If the address is in a translated region with a page size of 4 KB, then address bits [11:0] are untranslated but address bits [31:12] will be mapped and for these bits the virtual and physical addresses may be different. In this example, bit [12] could pose a potential problem due to virtual aliasing. Imagine two virtual addresses, VA0 and VA1, whose only difference is the value of bit [12], which map to the same physical address. These two virtual addresses would be indexed to two different lines by the cache, even though they were intended to represent the same physical address. Then if a program does a load using VA0 and a store using VA1, or vice-versa, the cache may not return the expected data.

Table 7-4 shows the overlapped virtual/physical address bits which could potentially be involved in virtual aliasing, given the possible minimum page sizes and cache way sizes supported by a 4KE core. Virtual aliasing is generally only a problem for the D-cache, since stores don't happen to the I-cache. No special hardware mechanism is provided to prevent the possibility of virtual aliasing, so it must be handled by software. The software solution must ensure that the

mapping of virtual address bits which overlap with physical address bits be handled consistently. The simplest approach is to ensure that the overlapping bits are unity-mapped (VA equals PA).

Table 7-4 Potential Virtual Aliasing Bits

Minimum Page Size (KB)	Cache Way Size (KB)	Overlapped address bits with possible aliasing
1	2	[10]
	4	[11:10]
	8	[12:10]
	16	[13:10]
4	8	[12]
	16	[13:12]
8	16	[13]

A related issue can occur in virtually indexed, physically tagged caches if the number of physical bits stored in the tag array do not fully overlap the physically translated bits for the smallest page size. For a 4KE core, there are always 22 address bits stored in the cache tag, representing bits [31:10] of the physical address. Since the minimum page size is 1 KB for the 4KEc, with bits [31:10] physically translated by the TLB, the cache tag size does overlap the translated bits and this issue will not occur.

7.3 Instruction Cache

The instruction cache (I-cache) is an optional on-chip memory block of up to 64 KB. The virtually indexed, physically tagged cache allows the virtual-to-physical address translation to occur in parallel with the cache access rather than having to wait for the physical address translation.

The core supports instruction cache locking. Cache locking allows critical code or data segments to be locked into the cache on a “per-line” basis, enabling the system programmer to maximize the efficiency of the system cache.

The cache locking function is always enabled on all instruction cache entries. Entries can then be marked as locked or unlocked on a per entry basis using the CACHE instruction.

7.4 Data Cache

The data cache (D-cache) is an optional on-chip memory block of up to 64 KB. The virtually indexed, physically tagged cache allows the virtual-to-physical address translation to occur in parallel with the cache access rather than having to wait for the physical address translation.

The core also supports a data cache locking mechanism identical to the instruction cache. Critical data segments to be locked into the cache on a “per-line” basis. The locked contents can be updated on a store hit, but cannot be selected for replacement on a miss.

The cache locking function is always enabled on all data cache entries. Entries can then be marked as locked or unlocked on a per entry basis using the CACHE instruction.

7.5 CACHE Instruction

Both caches support the CACHE instructions, which allow users to manipulate the contents of the Data and Tag arrays, including the locking of individual cache lines. Note that before the CACHE instructions are allowed to execute, all initiated refills are completed and stores are sent to the write buffer. The CACHE instructions are described in detail in Chapter 11, “MIPS32™ 4KE™ Processor Core Instructions,” on page 237.

The CACHE Index Load Tag and Index Store Tag instructions can be used to read and write the WS- RAM by setting the *WST* bit in the *ErrCtl* register. (The *ErrCtl* register is described in Section 5.2.34, "ErrCtl Register (CP0 Register 26, Select 0)" on page 144.) Note that when the *WST* bit is zero, the CACHE index instructions access the cache Tag array.

Not all values of the WS field are valid for defining the order in which the ways are selected. This is only an issue, however, if the WS-RAM is written after the initialization (invalidation) of the Tag array. Valid WS field encodings for way selection order is shown in Table 7-5, Table 7-6, and Table 7-7.

Table 7-5 Way Selection Encoding, 4 Ways

Selection Order[1]	WS[5:0]	Selection Order	WS[5:0]
0123	000000	2013	100010
0132	000001	2031	110010
0213	000010	2103	100110
0231	010010	2130	101110
0312	010001	2301	111010
0321	010011	2310	111110
1023	000100	3012	011001
1032	000101	3021	011011
1203	100100	3102	011101
1230	101100	3120	111101
1302	001101	3201	111011
1320	101101	3210	111111

1. The order is indicated by listing the least-recently used way to the left and the most-recently used way to the right, etc.

Table 7-6 Way Selection Encoding, 3 Ways

Selection Order[1]	WS[5:0][2]	Selection Order	WS[5:0]
012	0xx00x	120	1xx10x
021	0xx01x	201	1xx01x
102	0xx10x	210	1xx11x

1. The order is indicated by listing the least-recently used way to the left and the most-recently used way to the right, etc.
2. A “?” indicates a don’t care when written and unpredictable when read.

Table 7-7 Way Selection Encoding, 2 Ways

Selection Order[1]	WS[5:0][2]	Selection Order	WS[5:0]
01	xxx0xx	10	xxx1xx

1. The order is indicated by listing the least-recently used way to the left and the most-recently used way to the right, etc.
2. A “?” indicates a don’t care when written and unpredictable when read.

7.6 Software Cache Testing

Typically, the cache RAM arrays will be tested using BIST. It is, however, possible for software running on the processor to test all of the arrays. Of course, testing of the I-cache arrays should be done from an uncacheable space with interrupts disabled in order to maintain the cache contents. There are multiple methods for testing these arrays in software, only one is presented here.

7.6.1 I-Cache/D-cache Tag Arrays

These arrays can be tested via the Index Load Tag and Index Store Tag varieties of the CACHE instruction. Index Store Tag will write the contents of the *TagLo* register into the selected tag entry. Index Load Tag will read the selected tag entry into the *TagLo*.

7.6.2 I-Cache Data Array

This array can be tested using the Index Invalidate, Fill, and Index Load Tag varieties of the CACHE instruction. Fill will force a refill of the I-cache with data from a given address. In order to predict where the Fill data will go, it is advisable to invalidate the I-cache array prior to filling it. The last way invalidated will be the first way selected for replacement. Index Load Tag will read the selected data word into the *DataLo* register. The entire I-cache may be flushed using Index Invalidate. Then a test pattern can be stored into memory and the CACHE Fill operation will force the test pattern into the I-cache data array. Index Load Tags can be used to walk through each word of the I-cache array, checking the contents of the *DataLo* register against the expected value.

7.6.3 I-Cache WS Array

The testing of this array is very similar to the testing of the tag array. By setting the WST bit in the ErrCtl register, Index Load Tag and Index Store Tag CACHE instructions will read and write the WS array instead of the tag array.

7.6.4 D-Cache Data Array

This array can be tested using the Index Store Tag CACHE, SW, and LW instructions. First, use Index Store Tag to set the initial state of the tags to valid with a known physical address (PA). Write the array using SW instructions to the PAs that are resident in the cache. The value can then be read using LW instructions and compared to the expected data.

7.6.5 D-cache WS Array

The dirty bits in this array will be tested when the data tag is tested. The LRU bits can be tested using the same mechanism as the I-cache WS array.

7.7 Memory Coherence Issues

A cache presents coherency issues within the memory hierarchy which must be considered in the system design. Since a cache holds a copy of memory data, it is possible for another memory master to modify a memory location, thus making other copies of that location stale if those copies are still in use. A detailed discussion of memory coherence is beyond the scope of this document, but following are a few related comments.

A 4KE processor contains no direct hardware support for managing coherency with respect to its caches, so it must be handled via system design or software. The 4KE data cache supports either write-back or write-through protocols.

In write-through mode, all data writes will eventually be sent to memory. Due to write buffers, however, there could be a delay in how long it takes for the write to memory to actually occur. If another memory master updates cacheable memory which could also be in the 4KE caches, then those locations may need to be flushed from the cache. The only way to accomplish this invalidation is by use of the CACHE instruction.

In write-back mode, data writes only go to the cache and not to memory. So the processor cache may contain the *only* copy of data in the system until that data is written to main memory. Dirty lines are only written to memory when displaced from the cache as a new line is filled or if explicitly forced by certain flavors of the CACHE or PREF instructions.

The SYNC instruction may also be useful to software enforcing memory coherence, as it flushes the 4KE core’s write buffers.

Power Management

A MIPS32™ 4KE™ processor core offers a number of power management features, including low-power design, active power management and power-down modes of operation. The core is a static design that supports a WAIT instruction designed to signal the rest of the device that execution and clocking should be halted, reducing system power consumption during idle periods.

The core provides two mechanisms for system level low-power support discussed in the following sections.

- Section 8.1, "Register-Controlled Power Management"
- Section 8.2, "Instruction-Controlled Power Management"

8.1 Register-Controlled Power Management

The RP bit in the CP0 *Status* register enables a standard software mechanism for placing the system into a low power state. The state of the RP bit is available externally via the *SI_RP* output signal. Three additional pins, *SI_EX*L, *SI_ERL*, and *EJ_DebugM* support the power management function by allowing the user to change the power state if an exception or error occurs while the core is in a low power state.

Setting the RP bit of the CP0 *Status* register causes the core to assert the *SI_RP* signal. The external agent can then decide whether to reduce the clock frequency and place the core into power down mode.

If an interrupt is taken while the device is in power down mode, that interrupt may need to be serviced depending on the needs of the application. The interrupt causes an exception which in turn causes the EXL bit to be set. The setting of the EXL bit causes the assertion of the *SI_EXL* signal on the external bus, indicating to the external agent that an interrupt has occurred. At this time the external agent can choose to either speed up the clocks and service the interrupt or let it be serviced at the lower clock speed.

The setting of the ERL bit causes the assertion of the *SI_ERL* signal on the external bus, indicating to the external agent that an error has occurred. At this time the external agent can choose to either speed up the clocks and service the error or let it be serviced at the lower clock speed.

Similarly, the *EJ_DebugM* signal indicates that the processor is in debug mode. Debug mode is entered when the processor takes a debug exception. If fast handling of this is desired, the external agent can speed up the clocks.

The core provides four power down signals that are part of the system interface. Three of the pins change state as the corresponding bits in the CP0 *Status* register are set or cleared. The fourth pin indicates that the processor is in debug mode:

- The *SI_RP* signal represents the state of the RP bit (27) in the CP0 *Status* register.
- The *SI_EXL* signal represents the state of the EXL bit (1) in the CP0 *Status* register.
- The *SI_ERL* signal represents the state of the ERL bit (2) in the CP0 *Status* register.
- The *EJ_DebugM* signal indicates that the processor has entered debug mode.

8.2 Instruction-Controlled Power Management

The second mechanism for invoking power down mode is through execution of the WAIT instruction. If the bus is idle at the time the WAIT instruction reaches the M stage of the pipeline the internal clocks are suspended and the pipeline is frozen. However, the internal timer and some of the input pins (*SI_Int*[5:0], *SI_NMI*, *SI_Reset*, *SI_ColdReset*, and *EJ_DINT*) continue to run. If the bus is not idle at the time the WAIT instruction reaches the M stage, the pipeline stalls until the bus becomes idle, at which time the clocks are stopped. Once the CPU is in instruction controlled power management mode, any enabled interrupt, NMI, debug interrupt, or reset condition causes the CPU to exit this mode and resume normal operation. While the part is in this low-power mode, the *SI_SLEEP* signal is asserted to indicate to external agents what the state of the chip is.

EJTAG Debug Support

The EJTAG debug logic in the MIPS32™ 4KE™ processor cores provide three optional modules:

1. Hardware breakpoints
2. Test Access Port (TAP) for a dedicated connection to a debug host
3. EJTAG Trace for program counter/data address/data value trace to On-chip memory or to Trace probe.

This chapter contains the following sections:

9.1 Debug Control Register

The Debug Control Register (*DCR*) register controls and provides information about debug issues, and is always provided with the CPU core. The register is memory-mapped in drseg at offset 0x0.

The DataBrk and InstBrk bits indicate if hardware breakpoints are included in the implementation, and debug software is expected to read hardware breakpoint registers for additional information.

Hardware and software interrupts are maskable for non-debug mode with the INTE bit, which works in addition to the other mechanisms for interrupt masking and enabling. NMI is maskable in non-debug mode with the NMIE bit, and a pending NMI is indicated through the NMIP bit.

The SRE bit allows implementation dependent masking of none, some or all sources for soft reset. The soft reset masking may only be applied to a soft reset source if that source can be efficiently masked in the system, thus resulting in no reset at all. If that is not possible, then that soft reset source should not be masked, since a partial soft reset may cause the system to fail or hang. There is no automatic indication of whether the SRE is effective, so the user must consult system documentation.

The PE bit reflects the ProbEn bit from the EJTAG Control register (*ECR*), whereby the probe can indicate to the debug software running on the CPU if the probe expects to service dmseg accesses. The reset value in the table below takes effect on both hard and soft resets.

Debug Control Register

31 30	29	28 18	17	16	15 5	4	3	2	1	0
Res	ENM	Res	DB	IB	Res	INTE	NMIE	NMIP	SRE	PE

Table 9-1 *Debug Control Register* Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
Res	31:30	Reserved	R	0
ENM	29	Endianess in Kernel and Debug mode. 0: Little Endian 1: Big Endian	R	Preset
Res	28:18	Reserved	R	0
DB	17	Data Break Implemented. 0: No Data Break feature implemented 1: Data Break feature is implemented	R	Preset
IB	16	Instruction Break Implemented. 0: No Instruction Break feature implemented 1: Instruction Break feature is implemented	R	Preset
Res	15:5	Reserved	R	0

Table 9-1 *Debug Control Register* Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
INTE	4	Interrupt Enable in Normal Mode. This bit provides the hardware and software interrupt enable for non-debug mode, in addition to other masking mechanisms: 0: Interrupts disabled. 1: Interrupts enabled (depending on other enabling mechanisms).	R/W	1
NMIE	3	Non-Maskable Interrupt Enable for non-debug mode 0: NMI disabled. 1: NMI enabled.	R/W	1
NMIP	2	NMI Pending Indication. 0: No NMI pending. 1: NMI pending.	R	0
SRE	1	Soft Reset Enable This bit allows the system to mask soft resets. The core does not internally mask soft resets. Rather the state of this bit appears on the *EJ_SRstE* external output signal, allowing the system to mask soft resets if desired.	R/W	1
PE	0	Probe Enable This bit reflects the ProbEn bit in the EJTAG Control register. 0: No accesses to dmseg allowed 1: EJTAG probe services accesses to dmseg	R	Same value as ProbEn in ECR (see Table 9-4)

9.2 Hardware Breakpoints

Hardware breakpoints provide for the comparison by hardware of executed instructions and data load/store transactions. It is possible to set instruction breakpoints on addresses even in ROM area,. Data breakpoints can be set to cause a debug exception on a specific data transaction. Instruction and data hardware breakpoints are alike for many aspects, and are thus described in parallel in the following. The term hardware is not applied to breakpoint, unless required to distinguish it from software breakpoint.

There are two types of simple hardware breakpoints implemented in the 4KE cores; Instruction breakpoints and Data breakpoints.

A core may be configured with the following breakpoint options:

- No data or instruction breakpoints
- Two instruction and one data breakpoint
- Four instruction and two data breakpoints

9.2.1 Features of Instruction Breakpoint

Instruction breaks occur on instruction fetch operations and the break is set on the virtual address on the bus between the CPU and the instruction cache. Instruction breaks can also be made on the ASID value used by the MMU. Finally, a mask can be applied to the virtual address to set breakpoints on a range of instructions.

Instruction breakpoints compare the virtual address of the executed instructions (PC) and the ASID with the registers for each instruction breakpoint including masking of address and ASID. When an instruction breakpoint matches, a debug exception and/or a trigger is generated. An internal bit in the instruction breakpoint registers is set to indicate that the match occurred.

9.2.2 Features of Data Breakpoint

Data breakpoints occur on load/store transactions. Breakpoints are set on virtual address and ASID values, similar to the Instruction breakpoint. Data breakpoints can be set on a load, a store or both. Data breakpoints can also be set based on the value of the load/store operation. Finally, masks can be applied to both the virtual address and the load/store value.

Data breakpoints compare the transaction type (TYPE), which may be load or store, the virtual address of the transaction (ADDR), the ASID, accessed bytes (BYTELANE) and data value (DATA), with the registers for each data breakpoint including masking or qualification on the transaction properties. When a data breakpoint matches, a debug exception and/or a trigger is generated, and an internal bit in the data breakpoint registers is set to indicate that the match occurred. The match is precise in that the debug exception or trigger occurs on the instruction that caused the breakpoint to match.

9.2.3 Instruction Breakpoint Registers Overview

The register with implementation indication and status for instruction breakpoints in general is shown in Table 9-2.

Table 9-2 Overview of Status Register for Instruction Breakpoints

Register Mnemonic	Register Name and Description
IBS	Instruction Breakpoint Status

The four instruction breakpoints are numbered 0 to 3 for registers and breakpoints, and the number is indicated by n. The registers for each breakpoint are shown in Table 9-3.

Table 9-3 Overview of Registers for Each Instruction Breakpoint

Register Mnemonic	Register Name and Description
IBAn	Instruction Breakpoint Address n
IBMn	Instruction Breakpoint Address Mask n
IBASIDn	Instruction Breakpoint ASID n
IBCn	Instruction Breakpoint Control n

9.2.4 Data Breakpoint Registers Overview

The register with implementation indication and status for data breakpoints in general is shown in Table 9-4.

Table 9-4 Overview of Status Register for Data Breakpoints

Register Mnemonic	Register Name and Description
DBS	Data Breakpoint Status

The two data breakpoints are numbered 0 and 1 for registers and breakpoints, and the number is indicated by n. The registers for each breakpoint are shown in Table 9-5.

Table 9-5 Overview of Registers for each Data Breakpoint

Register Mnemonic	Register Name and Description
DBAn	Data Breakpoint Address n
DBMn	Data Breakpoint Address Mask n
DBASIDn	Data Breakpoint ASID n
DBCn	Data Breakpoint Control n
DBVn	Data Breakpoint Value n

9.2.5 Conditions for Matching Breakpoints

A number of conditions must be fulfilled in order for a breakpoint to match on an executed instruction or a data transaction, and the conditions for matching instruction and data breakpoints are described below. The breakpoints only match for instructions executed in non-debug mode, thus never on instructions executed in debug mode.

The match of an enabled breakpoint can either generate a debug exception or a trigger indication. The BE and/or TE bits in the *IBCn* or *DBCn* registers are used to enable the breakpoints.

Debug software should not configure breakpoints to compare on an ASID value unless a TLB is present in the implementation.

9.2.5.1 Conditions for Matching Instruction Breakpoints

When an instruction breakpoint is enabled, that breakpoint is evaluated for the address of every executed instruction in non-debug mode, including execution of instructions at an address causing an address error on an instruction fetch. The breakpoint is not evaluated on instructions from a speculative fetch or execution, nor for addresses which are unaligned with an executed instruction.

A breakpoint match depends on the virtual address of the executed instruction (PC) which can be masked at bit level, and match also can include an optional compare of ASID value. The registers for each instruction breakpoint have the values and mask used in the compare, and the equation that determines the match is shown below in C-like notation.

```
IB_match =
          ( ! IBCn_ASIDuse || ( ASID == IBASIDn_ASID ) ) &&
          ( <all 1's> == ( IBMn_IBM | ~ ( PC ^ IBAn_IBA ) ) )
```

The match indication for instruction breakpoints is always precise, i.e. indicated on the instruction causing the IB_match to be true.

9.2.5.2 Conditions for Matching Data Breakpoints

When a data breakpoint is enabled, that breakpoint is evaluated for every data transaction due to a load/store instruction executed in non-debug mode, including load/store for coprocessor, and transactions causing an address error on data access. The breakpoint is not evaluated due to a PREF instruction or other transactions which are not part of explicit load/store transactions in the execution flow, nor for addresses which are not the explicit load/store source or destination address.

A breakpoint match depends on the transaction type (TYPE) as load or store, the address, and optionally the data value of a transaction. The registers for each data breakpoint have the values and mask used in the compare, and the equation that determines the match is shown below in C-like notation.

The overall match equation is the DB_match.

```
DB_match =
          ( ( ( TYPE == load ) && ! DBCn_NoLB ) ||
            ( ( TYPE == store ) && ! DBCn_NoSB ) ) &&
          DB_addr_match && ( DB_no_value_compare || DB_value_match )
```

The match on the address part, DB_addr_match, depends on the virtual address of the transaction (ADDR), the ASID value, and the accessed bytes (BYTELANE) where BYTELANE[0] is 1 only if the byte at bits [7:0] on the bus is accessed, and BYTELANE[1] is 1 only if the byte at bits [15:8] is accessed, etc. The DB_addr_match is shown below.

```
DB_addr_match =
          ( ! DBCn_ASIDuse || ( ASID == DBASIDn_ASID ) ) &&
          ( <all 1's> == ( DBMn_DBM | ~ ( ADDR ^ DBAn_DBA ) ) ) &&
          ( <all 0's> != ( ~ BAI & BYTELANE ) )
```

The size of $DBCn_{\mathrm{BAI}}$ and BYTELANE is 4 bits.

Data value compare is included in the match condition for the data breakpoint depending on the bytes (BYTELANE as described above) accessed by the transaction, and the contents of breakpoint registers. The DB_no_value_compare is shown below.

```
DB_no_value_compare =
          ( <all 1's> == ( DBCn_BLM | DBCn_BAI | ~ BYTELANE ) )
```

The size of $DBCn_{\mathrm{BLM}}$, $DBCn_{\mathrm{BAI}}$ and BYTELANE is 4 bits.

In case a data value compare is required, DB_no_value_compare is false, then the data value from the data bus (DATA) is compared and masked with the registers for the data breakpoint. The endianess is not considered in these match equations for value, as the compare uses the data bus value directly, thus debug software is responsible for setup of the breakpoint corresponding with endianess.

$$
\begin{aligned}
&\texttt{DB\_value\_match} = \\
&\quad (\ (\ \texttt{DATA[7:0]} == \mathit{DBVn}_{\texttt{DBV[7:0]}}\ )\ ||\ !\ \texttt{BYTELANE[0]}\ ||\ \mathit{DBCn}_{\texttt{BLM[0]}}\ ||\ \mathit{DBCn}_{\texttt{BAI[0]}}\ )\ \&\& \\
&\quad (\ (\ \texttt{DATA[15:8]} == \mathit{DBVn}_{\texttt{DBV[15:8]}}\ )\ ||\ !\ \texttt{BYTELANE[1]}\ ||\ \mathit{DBCn}_{\texttt{BLM[1]}}\ ||\ \mathit{DBCn}_{\texttt{BAI[1]}}\ )\ \&\& \\
&\quad (\ (\ \texttt{DATA[23:16]} == \mathit{DBVn}_{\texttt{DBV[23:16]}}\ )\ ||\ !\ \texttt{BYTELANE[2]}\ ||\ \mathit{DBCn}_{\texttt{BLM[2]}}\ ||\ \mathit{DBCn}_{\texttt{BAI[2]}}\ )\&\& \\
&\quad (\ (\ \texttt{DATA[31:24]} == \mathit{DBVn}_{\texttt{DBV[31:24]}}\ )\ ||\ !\ \texttt{BYTELANE[3]}\ ||\ \mathit{DBCn}_{\texttt{BLM[3]}}\ ||\ \mathit{DBCn}_{\texttt{BAI[3]}}\ )
\end{aligned}
$$

The match for a data breakpoint is always precise, since the match expression is fully evaluated at the time the load/store instruction is executed. A true DB_match can thereby be indicated on the very same instruction causing the DB_match to be true.

9.2.6 Debug Exceptions from Breakpoints

Instruction and data breakpoints may be set up to generate a debug exception when the match condition is true, as described below.

9.2.6.1 Debug Exception by Instruction Breakpoint

If the breakpoint is enabled by BE bit in the *IBCn* register, then a debug instruction break exception occurs if the IB_match equation is true. The corresponding BS[n] bit in the *IBS* register is set when the breakpoint generates the debug exception.

The debug instruction break exception is always precise, so the *DEPC* register and DBD bit in the *Debug* register point to the instruction that caused the IB_match equation to be true.

The instruction receiving the debug exception does not update any registers due to the instruction, nor does any load or store by that instruction occur. Thus a debug exception from a data breakpoint can not occur for instructions receiving a debug instruction break exception.

The debug handler usually returns to the instruction causing the debug instruction break exception, whereby the instruction is executed. Debug software is responsible for disabling the breakpoint when returning to the instruction, otherwise the debug instruction break exception reoccurs.

9.2.6.2 Debug Exception by Data Breakpoint

If the breakpoint is enabled by BE bit in the *DBCn* register, then a debug exception occurs when the DB_match condition is true. The corresponding BS[n] bit in the *DBS* register is set when the breakpoint generates the debug exception.

A debug data break exception occurs when a data breakpoint indicates a match. In this case the *DEPC* register and DBD bit in the *Debug* register points to the instruction that caused the DB_match equation to be true.

The instruction causing the debug data break exception does not update any registers due to the instruction, and the following applies to the load or store transaction causing the debug exception:

- A store transaction is not allowed to complete the store to the memory system.
- A load transaction with no data value compare, i.e. where the DB_no_value_compare is true for the match, is not allowed to complete the load.
- A load transaction for a breakpoint with data value compare must occur from the memory system, since the value is required in order to evaluate the breakpoint.

The result of this is that the load or store instruction causing the debug data break exception appears as not executed, with the exception that a load from the memory system does occur for a breakpoint with data value compare, but the result of this load is discarded since the register file is not updated by the load.

If both data breakpoints without and with data value compare would match the same transaction and generate a debug exception, then the following rules apply with respect to updating the BS[n] bits.

- On both a load and store the BS[n] bits are required to be set for all matching breakpoints without a data value compare.
- On a store the BS[n] bits are allowed but not required to be set for all matching breakpoints with a data value compare, but either all or none of the BS[n] bits must be set for these breakpoints.
- On a load then no of the BS[n] bits are allowed to be set, since the load is not allowed to occur due to the debug exception from a breakpoint without a data value compare, and a valid data value is therefore not returned.

Any BS[n] bit set prior to the match and debug exception are kept set, since BS[n] bits are only cleared by debug software.

The debug handler usually returns to the instruction causing the debug data break exception, whereby the instruction is re-executed. This re-execution may result in a repeated load from system memory, since the load may have occurred previously in order to evaluate the breakpoint as described above. I/O devices with side effects on loads must be able to allow such reloads, or debug software should alternatively avoid setting data breakpoints with data value compares on such I/O devices. Debug software is responsible for disabling breakpoints when returning to the instruction, otherwise the debug data break exception will reoccur.

9.2.7 Breakpoint used as TriggerPoint

Both instruction and data hardware breakpoints can be setup by software so a matching breakpoint does not generate a debug exception, but only an indication through the BS[n] bit. The TE bit in the *IBCn* or *DBCn* register controls if an instruction or data breakpoint is used as a so-called triggerpoint. The triggerpoints are, like breakpoints, only compared for instructions executed in non-debug mode.

The BS[n] bit in the *IBS* or *DBS* register is set when the respective IB_match or DB_match bit is true.

The triggerpoint feature can be used to start and stop tracing. See Section 9.10, "EJTAG Trace Enabling" for details.

9.2.8 Instruction Breakpoint Registers

The registers for instruction breakpoints are described below. These registers have implementation information and are used to set up the instruction breakpoints. All registers are in drseg, and the addresses are shown in Table 9-6.

Table 9-6 Addresses for Instruction Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x1000	*IBS*	Instruction Breakpoint Status
0x1100 + n * 0x100	*IBAn*	Instruction Breakpoint Address n
0x1108 + n * 0x100	*IBMn*	Instruction Breakpoint Address Mask n
0x1110 + n * 0x100	*IBASIDn*	Instruction Breakpoint ASID n

Table 9-6 Addresses for Instruction Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x1118 + n * 0x100	*IBCn*	Instruction Breakpoint Control n
Note: n is breakpoint number in range 0 to 3 (or 0 to 1, depending on the implemented hardware)		

An example of some of the registers; *IBA0* is at offset 0x1100 and *IBC2* is at offset 0x1318.

9.2.8.1 Instruction Breakpoint Status (*IBS*) Register

Compliance Level: Implemented only if instruction breakpoints are implemented.

The Instruction Breakpoint Status (*IBS*) register holds implementation and status information about the instruction breakpoints.

The ASID applies to all the instruction breakpoints.

***IBS* Register Format**

31	30	29 28	27 24	23 4	3 0
Res	ASID sup	Res	BCN	Res	BS

Table 9-7 *IBS* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
Res	31	Must be written as zero; returns zero on read.	R	0
ASIDsup	30	Indicates that ASID compare is supported in instruction breakpoints. 0: No ASID compare. 1: ASID compare (IBASIDn register implemented). 1: Supported 0: Not supported	R	4KEc core - 1 4KEm/p cores - 0
Res	29:28	Must be written as zero; returns zero on read.	R	0
BCN	27:24	Number of instruction breakpoints implemented.	R	4 or 2[a]
Res	23:4	Must be written as zero; returns zero on read.	R	0
BS	3:0	Break status for breakpoint n is at BS[n], with n from 0 to 3[b]. The bit is set to 1 when the condition for the corresponding breakpoint has matched.	R/W	Undefined

Note: [a] Based on actual hardware implemented.

Note: [b] In case of only 2 Instruction breakpoints bit 2 and 3 become reserved.

9.2.8.2 Instruction Breakpoint Address n (*IBAn*) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

The Instruction Breakpoint Address n (*IBAn*) register has the address used in the condition for instruction breakpoint n

IBAn **Register Format**

31 … 0
IBA

Table 9-8 *IBAn* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	**Bit(s)**			
IBA	31:0	Instruction breakpoint address for condition.	R/W	Undefined

9.2.8.3 Instruction Breakpoint Address Mask n (*IBMn*) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

The Instruction Breakpoint Address Mask n (*IBMn*) register has the mask for the address compare used in the condition for instruction breakpoint n.

***IBMn* Register Format**

31 … 0
IBM

Table 9-9 *IBMn* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
IBM	31:0	Instruction breakpoint address mask for condition: 0: Corresponding address bit not masked. 1: Corresponding address bit masked.	R/W	Undefined

9.2.8.4 Instruction Breakpoint ASID n (*IBASIDn*) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

The Instruction Breakpoint ASID n (*IBASIDn*) register has the ASID value used in the compare for instruction breakpoint n. The number of bits in the ASID field is 8, to match the ASID size in the TLB. This register is only valid for the 4KEc core.

***IBASIDn* Register Format**

31 — 8	7 — 0
Res	ASID

Table 9-10 *IBASIDn* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
Res	31:8	Must be written as zero; returns zero on read.	R	0
ASID	7:0	Instruction breakpoint ASID value for a compare.	R/W	Undefined

9.2.8.5 Instruction Breakpoint Control n (*IBCn*) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

The Instruction Breakpoint Control n (*IBCn*) register controls the setup of instruction breakpoint n.

***IBCn* Register Format**

31 24	23	22 3	2	1	0
Res	ASID use	Res	TE	Res	BE

Table 9-11 *IBCn* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
Res	31:24	Must be written as zero; returns zero on read.	R	0
ASIDuse	23	Use ASID value in compare for instruction breakpoint n: 0: Don't use ASID value in compare 1: Use ASID value in compare	4KEc core - R/W 4KEm/4KEp cores - 0	Undefined
Res	22:3	Must be written as zero; returns zero on read.	R	0
TE	2	Use instruction breakpoint n as triggerpoint: 0: Don't use it as triggerpoint 1: Use it as triggerpoint	R/W	0
Res	1	Must be written as zero; returns zero on read.	R	0
BE	0	Use instruction breakpoint n as breakpoint: 0: Don't use it as breakpoint 1: Use it as breakpoint	R/W	0

9.2.9 Data Breakpoint Registers

The registers for data breakpoints are described below. These registers have implementation information and are used the setup the data breakpoints. All registers are in drseg, and the addresses are shown in Table 9-12.

Table 9-12 Addresses for Data Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x2000	*DBS*	Data Breakpoint Status
0x2100 + 0x100 * n	*DBAn*	Data Breakpoint Address n
0x2108 + 0x100 * n	*DBMn*	Data Breakpoint Address Mask n
0x2110 + 0x100 * n	*DBASIDn*	Data Breakpoint ASID n
0x2118 + 0x100 * n	*DBCn*	Data Breakpoint Control n
0x2120 + 0x100 * n	*DBVn*	Data Breakpoint Value n
Note: n is breakpoint number as 0 or 1 (or just 0, depending on the implemented hardware)		

An example of some of the registers; *DBM0* is at offset 0x2108 and *DBV1* is at offset 0x2220.

9.2.9.1 Data Breakpoint Status (*DBS*) Register

Compliance Level: Implemented if data breakpoints are implemented.

The Data Breakpoint Status (*DBS*) register holds implementation and status information about the data breakpoints.

The ASIDsup field indicates whether ASID compares are supported.

***DBS* Register Format**

31	30	29 28	27 24	23 2	1 0
Res	ASID sup	Res	BCN	Res	BS

Table 9-13 *DBS* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
Res	31	Must be written as zero; returns zero on read.	R	0
ASID	30	Indicates that ASID compares are supported in data breakpoints. 0: Not supported 1: Supported	R	4KEc core - 1 4KEm/p cores - 0
Res	29:28	Must be written as zero; returns zero on read.	R	0
BCN	27:24	Number of data breakpoints implemented.	R	2 or 1[a]
Res	23:2	Must be written as zero; returns zero on read.	R	0
BS	1:0	Break status for breakpoint n is at BS[n], with n from 0 to 1[b]. The bit is set to 1 when the condition for the corresponding breakpoint has matched.	R/W0	Undefined

Note: [a] Based on actual hardware implemented.

Note: [b] In case of only 1 data breakpoint bit 1 become reserved.

9.2.9.2 Data Breakpoint Address n (*DBAn*) Register

Compliance Level: Implemented only for implemented data breakpoints.

The Data Breakpoint Address n (*DBAn*) register has the address used in the condition for data breakpoint n.

DBAn **Register Format**

31 ... 0
DBA

Table 9-14 *DBAn* Register Field Descriptions

Fields				
Name	Bit(s)	Description	Read/ Write	Reset State
DBA	31:0	Data breakpoint address for condition.	R/W	Undefined

9.2.9.3 Data Breakpoint Address Mask n (*DBMn*) Register

Compliance Level: Implemented only for implemented data breakpoints.

The Data Breakpoint Address Mask n (*DBMn*) register has the mask for the address compare used in the condition for data breakpoint n.

DBMn **Register Format**

31 0
DBM

Table 9-15 *DBMn* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
DBM	31:0	Data breakpoint address mask for condition: 0: Corresponding address bit not masked 1: Corresponding address bit masked	R/W	Undefined

9.2.9.4 Data Breakpoint ASID n (*DBASIDn*) Register

Compliance Level: Implemented only for implemented data breakpoints.

The Data Breakpoint ASID n (*DBASIDn*) register has the ASID value used in the compare for data breakpoint n.

This register is only valid in the 4Kc core.

***DBASIDn* Register Format**

31 ... 8	7 ... 0
Res	ASID

Table 9-16 *DBASIDn* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
Res	31:8	Must be written as zero; returns zero on read.	R	0
ASID	7:0	Data breakpoint ASID value for compares.	R/W	Undefined

9.2.9.5 Data Breakpoint Control n (*DBCn*) Register

Compliance Level: Implemented only for implemented data breakpoints.

The Data Breakpoint Control n (*DBCn*) register controls the setup of data breakpoint n.

***DBCn* Register Format**

31 24	23	22 18	17 14	13	12	11 8	7 4	3	2	1	0
Re	ASID use	Res	BAI	NoSB	NoLB	Res	BLM	Res	TE	Res	BE

Table 9-17 *DBCn* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
Res	31:24	Must be written as zero; returns zero on reads.	R	0
ASIDuse	23	Use ASID value in compare for data breakpoint n: 0: Don’t use ASID value in compare 1: Use ASID value in compare	4Kc core - R/W 4Km/4Kp cores - 0	Undefined
Res	22:18	Must be written as zero; returns zero on reads.	R	0
BAI	17:14	Byte access ignore controls ignore of access to a specific byte. BAI[0] ignores access to byte at bits [7:0] of the data bus, BAI[1] ignores access to byte at bits [15:8], etc. 0: Condition depends on access to corresponding byte 1: Access for corresponding byte is ignored	R/W	Undefined
NoSB	13	Controls if condition for data breakpoint is not fulfilled on a store transaction: 0: Condition may be fulfilled on store transaction 1: Condition is never fulfilled on store transaction	R/W	Undefined
NoLB	12	Controls if condition for data breakpoint is not fulfilled on a load transaction: 0: Condition may be fulfilled on load transaction 1: Condition is never fulfilled on load transaction	R/W	Undefined
Res	11:8	Must be written as zero; returns zero on reads.	R	0
BLM	7:4	Byte lane mask for value compare on data breakpoint. BLM[0] masks byte at bits [7:0] of the data bus, BLM[1] masks byte at bits [15:8], etc.: 0: Compare corresponding byte lane 1: Mask corresponding byte lane	R/W	Undefined
Res	3	Must be written as zero; returns zero on reads.	R	0

Table 9-17 *DBCn* Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	**Bits**			
TE	2	Use data breakpoint n as triggerpoint: 0: Don’t use it as triggerpoint 1: Use it as triggerpoint	R/W	0
Res	1	Must be written as zero; returns zero on reads.	R	0
BE	0	Use data breakpoint n as breakpoint: 0: Don’t use it as breakpoint 1: Use it as breakpoint	R/W	0

9.2.9.6 Data Breakpoint Value n (*DBVn*) Register

Compliance Level: Implemented only for implemented data breakpoints.

The Data Breakpoint Value n (*DBVn*) register has the value used in the condition for data breakpoint n.

***DBVn* Register Format**

31 ... 0
DBV

Table 9-18 *DBVn* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
DBV	31:0	Data breakpoint value for condition.	R/W	Undefined

9.3 Test Access Port (TAP)

The following main features are supported by the TAP module:

- 5-pin industry standard JTAG Test Access Port (*TCK*, *TMS*, *TDI*, *TDO*, *TRST_N*) interface which is compatible with IEEE Std. 1149.1.
- Target chip and EJTAG feature identification available through the Test Access Port (TAP) controller.
- The processor can access external memory on the EJTAG Probe serially through the EJTAG pins. This is achieved through Processor Access (PA), and is used to eliminate the use of the system memory for debug routines.
- Support for both ROM based debugger and debugging both through TAP.

9.3.1 EJTAG Internal and External Interfaces

The external interface of the EJTAG module consists of the 5 signals defined by the IEEE standard.

Table 9-19 EJTAG Interface Pins

Pin	Type	Description
TCK	I	Test Clock Input Input clock used to shift data into or out of the Instruction or data registers. The *TCK* clock is independent of the processor clock, so the EJTAG probe can drive *TCK* independently of the processor clock frequency. The core signal for this is called *EJ_TCK*
TMS	I	Test Mode Select Input The *TMS* input signal is decoded by the TAP controller to control test operation. *TMS* is sampled on the rising edge of *TCK*. The core signal for this is called *EJ_TMS*
TDI	I	Test Data Input Serial input data (*TDI*) is shifted into the Instruction register or data registers on the rising edge of the *TCK* clock, depending on the TAP controller state. The core signal for this is called *EJ_TDI*
TDO	O	Test Data Output Serial output data is shifted from the Instruction or data register to the *TDO* pin on the falling edge of the *TCK* clock. When no data is shifted out, the *TDO* is 3-stated. The core signal for this is called *EJ_TDO* with output enable controlled by *EJ_TDOzstate*.

Table 9-19 EJTAG Interface Pins (Continued)

Pin	Type	Description
TRST_N	I	Test Reset Input (Optional pin) The *TRST_N* pin is an active-low signal for asynchronous reset of the TAP controller and instruction in the TAP module, independent of the processor logic. The processor is not reset by the assertion of *TRST_N*. The core signal for this is called *EJ_TRST_N* This signal is optional, but power-on reset must apply a low pulse on this signal at power-on and then leave it high, in case the signal is not available as a pin on the chip. If available on the chip, then it must be low on the board when the EJTAG debug features are unused by the probe.

9.3.2 Test Access Port Operation

The TAP controller is controlled by the Test Clock (*TCK*) and Test Mode Select (*TMS*) inputs. These two inputs determine whether an the Instruction register scan or data register scan is performed. The TAP consists of a small controller, driven by the *TCK* input, which responds to the *TMS* input as shown in the state diagram in Figure 9-1 on page 189. The TAP uses both clock edges of *TCK*. *TMS* and *TDI* are sampled on the rising edge of *TCK*, while *TDO* changes on the falling edge of *TCK*.

At power-up the TAP is forced into the *Test-Logic-Reset* by low value on *TRST_N*. The TAP instruction register is thereby reset to IDCODE. No other parts of the EJTAG hardware are reset through the *Test-Logic-Reset* state.

When test access is required, a protocol is applied via the *TMS* and *TCK* inputs, causing the TAP to exit the *Test-Logic-Reset* state and move through the appropriate states. From the *Run-Test/Idle* state, an Instruction register scan or a data register scan can be issued to transition the TAP through the appropriate states shown in Figure 9-1 on page 189.

The states of the data and instruction register scan blocks are mirror images of each other adding symmetry to the protocol sequences. The first action that occurs when either block is entered is a capture operation. For the data registers, the *Capture-DR* state is used to capture (or parallel load) the data into the selected serial data path. In the Instruction register, the *Capture-IR* state is used to capture status information into the Instruction register.

From the *Capture* states, the TAP transitions to either the *Shift* or *Exit1* states. Normally the *Shift* state follows the *Capture* state so that test data or status information can be shifted out for inspection and new data shifted in. Following the *Shift* state, the TAP either returns to the *Run-Test/Idle* state via the *Exit1* and *Update* states or enters the *Pause* state via *Exit1*. The reason for entering the *Pause* state is to temporarily suspend the shifting of data through either the Data or Instruction Register while a required operation, such as refilling a host memory buffer, is performed. From the Pause state shifting can resume by re-entering the *Shift* state via the *Exit2* state or terminate by entering the *Run-Test/Idle* state via the *Exit2* and *Update* states.

Upon entering the data or Instruction register scan blocks, shadow latches in the selected scan path are forced to hold their present state during the Capture and Shift operations. The data being shifted into the selected scan path is not output through the shadow latch until the TAP enters the *Update-DR* or *Update-IR* state. The *Update* state causes the shadow latches to update (or parallel load) with the new data that has been shifted into the selected scan path.

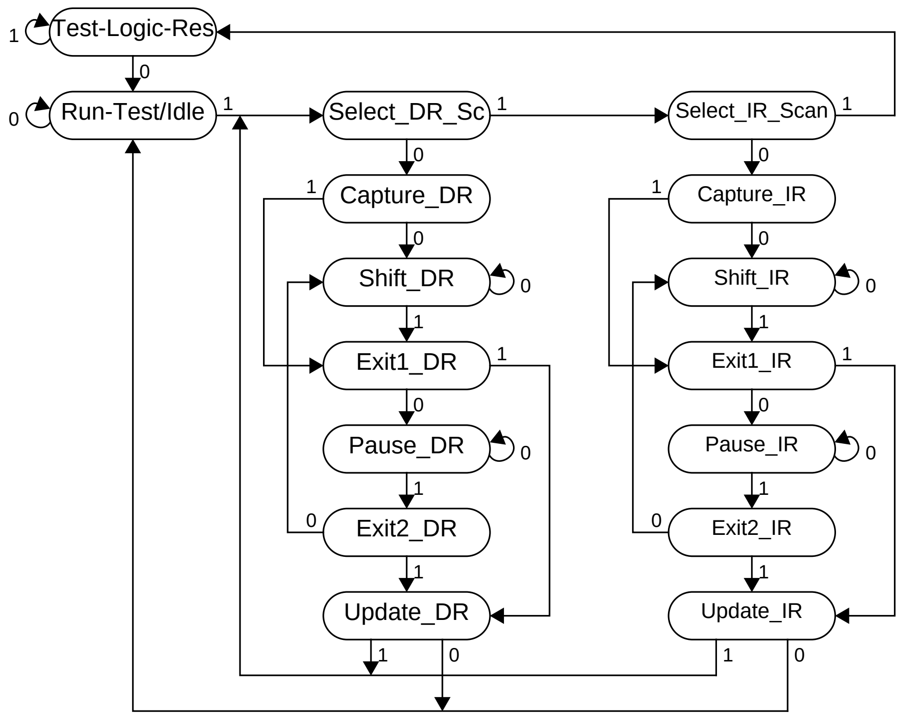


Figure 9-1 TAP Controller State Diagram

9.3.2.1 Test-Logic-Reset State

In the *Test-Logic-Reset* state the boundary scan test logic is disabled. The test logic enters the *Test-Logic-Reset* state when the *TMS* input is held HIGH for at least five rising edges of *TCK*. The BYPASS instruction is forced into the instruction register output latches during this state. The controller remains in the *Test-Logic-Reset* state as long as *TMS* is HIGH.

9.3.2.2 Run-Test/Idle State

The controller enters the *Run-Test/Idle* state between scan operations. The controller remains in this state as long as *TMS* is held LOW. The instruction register and all test data registers retain their previous state. The instruction cannot change when the TAP controller is in this state.

When *TMS* is sampled HIGH on the rising edge of *TCK*, the controller transitions to the *Select_DR* state.

9.3.2.3 Select_DR_Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, then the controller transitions to the *Capture_DR* state. A HIGH on *TMS* causes the controller to transition to the *Select_IR* state. The instruction cannot change while the TAP controller is in this state.

9.3.2.4 Select_IR_Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller transitions to the *Capture_IR* state. A HIGH on *TMS* causes the controller to transition to the *Test-Reset-Logic* state. The instruction cannot change while the TAP controller is in this state.

9.3.2.5 Capture_DR State

In this state the boundary scan register captures the value of the register addressed by the Instruction register, and the value is then shifted out in the *Shift_DR*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_DR* state. The instruction cannot change while the TAP controller is in this state.

9.3.2.6 Shift_DR State

In this state the test data register connected between *TDI* and *TDO* as a result of the current instruction shifts data one stage toward its serial output on the rising edge of *TCK*. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller remains in the *Shift_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_DR* state. The instruction cannot change while the TAP controller is in this state.

9.3.2.7 Exit1_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Pause_DR* state. A HIGH on *TMS* causes the controller to transition to the *Update_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

9.3.2.8 Pause_DR State

The *Pause_DR* state allows the controller to temporarily halt the shifting of data through the test data register in the serial path between *TDI* and *TDO*. All test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller remains in the *Pause_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit2_DR* state. The instruction cannot change while the TAP controller is in this state.

9.3.2.9 Exit2_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift_DR* state to allow another serial shift of data. A HIGH on *TMS* causes the controller to transition to the *Update_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

9.3.2.10 Update_DR State

When the TAP controller is in this state the value shifted in during the *Shift_DR* state takes effect on the rising edge of the *TCK* for the register indicated by the Instruction register.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Run-Test/Idle* state. A HIGH on *TMS* causes the controller to transition to the *Select_DR_Scan* state. The instruction cannot change while the TAP controller is in this state and all shift register stages in the test data registers selected by the current instruction retain their previous state.

9.3.2.11 Capture_IR State

In this state the shift register contained in the Instruction register loads a fixed pattern (00001_2) on the rising edge of *TCK*. The data registers selected by the current instruction retain their previous state.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_IR* state. The instruction cannot change while the TAP controller is in this state.

9.3.2.12 Shift_IR State

In this state the instruction register is connected between *TDI* and *TDO* and shifts data one stage toward its serial output on the rising edge of *TCK*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller remains in the *Shift_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1_IR* state.

9.3.2.13 Exit1_IR State

This is a temporary controller state in which all registers retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Pause_IR* state. A HIGH on *TMS* causes the controller to transition to the *Update_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state and the instruction register retains its previous state.

9.3.2.14 Pause_IR State

The *Pause_IR* state allows the controller to temporarily halt the shifting of data through the instruction register in the serial path between *TDI* and *TDO*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller remains in the *Pause_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit2_IR* state. The instruction cannot change while the TAP controller is in this state.

9.3.2.15 Exit2_IR State

This is a temporary controller state in which the instruction register retains its previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, then the controller transitions to the *Shift_IR* state to allow another serial shift of data. A HIGH on *TMS* causes the controller to transition to the *Update_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

9.3.2.16 Update_IR State

The instruction shifted into the instruction register takes effect on the rising edge of *TCK*.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Run-Test/Idle* state. A HIGH on *TMS* causes the controller to transition to the *Select_DR_Scan* state.

9.3.3 Test Access Port (TAP) Instructions

The TAP Instruction register allows instructions to be serially input into the device when TAP controller is in the *Shift-IR* state. Instructions are decoded and define the serial test data register path that is used to shift data between *TDI* and *TDO* during data register scanning.

The Instruction register is a 5-bit register. In the current EJTAG implementation only some instructions have been decoded; the unused instructions default to the BYPASS instruction.

Table 9-20 Implemented EJTAG Instructions

Value	Instruction	Function
0x01	IDCODE	Select Chip Identification data register
0x03	IMPCODE	Select Implementation register
0x08	ADDRESS	Select Address register
0x09	DATA	Select Data register

Table 9-20 Implemented EJTAG Instructions (Continued)

Value	Instruction	Function
0x0A	CONTROL	Select EJTAG Control register
0x0B	ALL	Select the Address, Data and EJTAG Control registers
0x0C	EJTAGBOOT	Set EjtagBrk, ProbEn and ProbTrap to 1 as reset value
0x0D	NORMALBOOT	Set EjtagBrk, ProbEn and ProbTrap to 0 as reset value
0x0E	FASTDATA	Selects the Data and Fastdata registers
0x10	TCBCONTROLA	Selects the *TCBTCONTROLA* register in the Trace Control Block
0x11	TCBCONTROLB	Selects the *TCBTCONTROLB* register in the Trace Control Block
0x12	TCBDATA	Selects the *TCBDATA* register in the Trace Control Block
0x1F	BYPASS	Bypass mode

9.3.3.1 BYPASS Instruction

The required BYPASS instruction allows the processor to remain in a functional mode and selects the Bypass register to be connected between *TDI* and *TDO*. The BYPASS instruction allows serial data to be transferred through the processor from *TDI* to *TDO* without affecting its operation. The bit code of this instruction is defined to be all ones by the IEEE 1149.1 standard. Any unused instruction is defaulted to the BYPASS instruction.

9.3.3.2 IDCODE Instruction

The IDCODE instruction allows the processor to remain in its functional mode and selects the Device Identification (ID) register to be connected between *TDI* and *TDO*. The Device ID register is a 32-bit shift register containing information regarding the IC manufacturer, device type, and version code. Accessing the Identification Register does not interfere with the operation of the processor. Also, access to the Identification Register is immediately available, via a TAP data scan operation, after power-up when the TAP has been reset with on-chip power-on or through the optional *TRST_N* pin.

9.3.3.3 IMPCODE Instruction

This instruction selects the Implementation register for output, which is always 32 bits.

9.3.3.4 ADDRESS Instruction

This instruction is used to select the Address register to be connected between *TDI* and *TDO*. The EJTAG Probe shifts 32 bits through the *TDI* pin into the Address register and shifts out the captured address via the *TDO* pin.

9.3.3.5 DATA Instruction

This instruction is used to select the Data register to be connected between *TDI* and *TDO*. The EJTAG Probe shifts 32 bits of *TDI* data into the Data register and shifts out the captured data via the *TDO* pin.

9.3.3.6 CONTROL Instruction

This instruction is used to select the EJTAG Control register to be connected between *TDI* and *TDO*. The EJTAG Probe shifts 32 bits of *TDI* data into the EJTAG Control register and shifts out the EJTAG Control register bits via *TDO*.

9.3.3.7 ALL Instruction

This instruction is used to select the concatenation of the Address and Data register, and the EJTAG Control register between *TDI* and *TDO*. It can be used in particular if switching instructions in the instruction register takes too many *TCK* cycles. The first bit shifted out is bit 0.

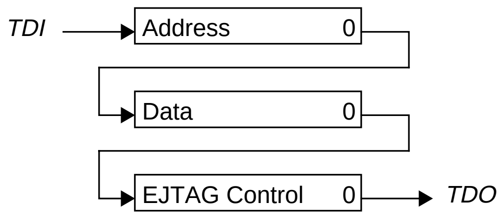


Figure 9-2 Concatenation of the EJTAG Address, Data and Control Registers

9.3.3.8 EJTAGBOOT Instruction

When the EJTAGBOOT instruction is given and the Update-IR state is left, then the reset values of the ProbTrap, ProbEn and EjtagBrk bits in the EJTAG Control register are set to 1 after a hard or soft reset.

This EJTAGBOOT indication is effective until a NORMALBOOT instruction is given, *TRST_N* is asserted or a rising edge of *TCK* occurs when the TAP controller is in Test-Logic-Reset state.

It is possible to make the CPU go into debug mode just after a hard or soft reset, without fetching or executing any instructions from the normal memory area. This can be used for download of code to a system which have no code in ROM.

The Bypass register is selected when the EJTAGBOOT instruction is given.

9.3.3.9 NORMALBOOT Instruction

When the NORMALBOOT instruction is given and the Update-IR state is left, then the reset value of the ProbTrap, ProbEn and EjtagBrk bits in the EJTAG Control register are set to 0 after hard or soft reset.

The Bypass register is selected when the NORMALBOOT instruction is given.

9.3.3.10 FASTDATA Instruction

This selects the Data and the Fastdata registers at once, as shown in Figure 9-3.

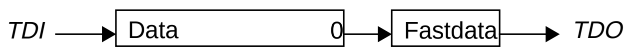


Figure 9-3 TDI to TDO Path when in Shift-DR State and FASTDATA Instruction is Selected

9.3.3.11 TCBCONTROLA Instruction

This instruction is used to select the TCBCONTROLA register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

9.3.3.12 TCBCONTROLB Instruction

This instruction is used to select the TCBCONTROLB register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

9.3.3.13 TCBDATA Instruction

This instruction is used to select the TCBDATA register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register. It should be noted that the TCBDATA register is only an access register to other TCB registers. The width of the TCBDATA register is dependent on the specific TCB register.

9.4 EJTAG TAP Registers

The EJTAG TAP Module has one Instruction register and a number of data registers, all accessible through the TAP:

9.4.1 Instruction Register

The Instruction register is accessed when the TAP receives an Instruction register scan protocol. During an Instruction register scan operation the TAP controller selects the output of the Instruction register to drive the *TDO* pin. The shift register consists of a series of bits arranged to form a single scan path between *TDI* and *TDO*. During an Instruction register scan operations, the TAP controls the register to capture status information and shift data from *TDI* to *TDO*. Both the capture and shift operations occur on the rising edge of *TCK*. However, the data shifted out from the *TDO* occurs on the falling edge of *TCK*. In the Test-Logic-Reset and *Capture-IR* state, the instruction shift register is set to 00001_2, as for the IDCODE instruction. This forces the device into the functional mode and selects the Device ID register. The Instruction register is 5 bits wide. The instruction shifted in takes effect for the following data register scan operation. A list of the implemented instructions are listed in Table 9-20.

9.4.2 Data Registers Overview

The EJTAG uses several data registers, which are arranged in parallel from the primary *TDI* input to the primary *TDO* output. The Instruction register supplies the address that allows one of the data registers to be accessed during a data register scan operation. During a data register scan operation, the addressed scan register receives TAP control signals to capture the register and shift data from *TDI* to *TDO*. During a data register scan operation, the TAP selects the output of the data register to drive the *TDO* pin. The register is updated in the *Update-DR* state with respect to the write bits.

This description applies in general to the following data registers:

- Bypass Register
- Device Identification Register
- Implementation Register
- EJTAG Control Register (ECR)
- Processor Access Address Register
- Processor Access Data Register
- FastData Register

9.4.2.1 Bypass Register

The *Bypass* register consists of a single scan register bit. When selected, the Bypass register provides a single bit scan path between *TDI* and *TDO*. The Bypass register allows abbreviating the scan path through devices that are not involved in the test. The Bypass register is selected when the Instruction register is loaded with a pattern of all ones to satisfy the IEEE 1149.1 Bypass instruction requirement.

9.4.2.2 Device Identification (*ID*) Register

The *Device Identification* register is defined by IEEE 1149.1, to identify the device's manufacturer, part number, revision, and other device-specific information. Table 9-21 shows the bit assignments defined for the read-only Device Identification Register, and inputs to the core determine the value of these bits. These bits can be scanned out of the *ID* register after being selected. The register is selected when the Instruction register is loaded with the IDCODE instruction.

Device Identification **Register Format**

31 28	27 12	11 1	0
Version	PartNumber	ManufID	R

Table 9-21 Device Identification Register

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
Version	31:28	**Version** (4 bits) This field identifies the version number of the processor derivative.	R	*EJ_Version[3:0]*
PartNumber	27:12	**Part Number** (16 bits) This field identifies the part number of the processor derivative.	R	*EJ_PartNumber[15:0]*
ManufID	11:1	**Manufacturer Identity** (11 bits) Accordingly to IEEE 1149.1-1990, the manufacturer identity code shall be a compressed form of the JEDEC Publications 106-A.	R	*EJ_ManufID[10:0]*
R	0	reserved	R	1

9.4.2.3 *Implementation* Register

This 32-bit read-only register is used to identify the features of the EJTAG implementation. Some of the reset values are set by inputs to the core. The register is selected when the Instruction register is loaded with the IMPCODE instruction.

Implementation **Register Format**

31 29	28 25	24	23 21	20 17	16	15	14	13 0
EJTAGver	reserved	DINTsup	ASIDsize	reserved	MIPS16	0	NoDMA	reserved

Table 9-22 *Implementation* Register Descriptions

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
EJTAGver	31:29	EJTAG Version. 2: Version 2.6	R	2
reserved	28:25	reserved	R	0
DINTsup	24	DINT Signal Supported from Probe This bit indicates if the DINT signal from the probe is supported: 0: DINT signal from the probe is not supported 1: Probe can use DINT signal to make debug interrupt.	R	*EJ_DINTsup*
ASIDsize	23:21	Size of ASID field in implementation: 0: No ASID in implementation 1: 6-bit ASID 2: 8-bit ASID 3: Reserved	R	4KEc core - 2 4KEm/4KEp cores - 0
reserved	20:17	reserved	R	0
MIPS16	16	Indicates whether MIPS16 is implemented 0: No MIPS16 support 1: MIPS16 implemented	R	Preset
reserved	15	reserved	R	0
NoDMA	14	No EJTAG DMA Support	R	1
reserved	13:0	reserved	R	0

9.4.2.4 EJTAG Control Register

This 32-bit register controls the various operations of the TAP modules. This register is selected by shifting in the CONTROL instruction. Bits in the EJTAG Control register can be set/cleared by shifting in data; status is read by shifting out the contents of this register. This EJTAG Control register can only be accessed by the TAP interface.

The EJTAG Control register is not updated in the *Update-DR* state unless the Reset occurred (Rocc) bit 31, is either 0 or written to 0. This is in order to ensure prober handling of processor accesses.

The value used for reset indicated in the table below takes effect on both hard and soft CPU resets, but not on TAP controller resets by e.g. *TRST_N*. *TCK* clock is not required when the hard or soft CPU reset occurs, but the bits are still updated to the reset value when the *TCK* applies. The first 5 *TCK* clocks after hard or soft CPU resets may result in reset of the bits, due to synchronization between clock domains.

***EJTAG Control* Register Format**

31	30 29	28 23	22	21	20	19	18	17	16	15	14	13	12	11 4	3	2 0
Rocc	Psz	Res	Doze	Halt	PerRst	PRnW	PrAcc	Res	PrRst	ProbEn	ProbTrap	Res	EjtagBrk	Res	DM	Res

Table 9-23 *EJTAG Control* Register Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>Rocc</td><td>31</td><td>Reset Occurred

The bit indicates if a hard or soft reset has occurred:
0: No reset occurred since bit last cleared.
1: Reset occurred since bit last cleared.

The Rocc bit will keep the 1 value as long as a hard or soft reset is applied.

This bit must be cleared by the probe, to acknowledge that the incident was detected.

The EJTAG Control register is not updated in the Update-DR state unless Rocc is 0, or written to 0. This is in order to ensure proper handling of processor access.</td><td>R/W</td><td>1</td></tr>
<tr><td>Psz[1:0]</td><td>30:29</td><td>Processor Access Transfer Size

These bits are used in combination with the lower two address bits of the Address register to determine the size of a processor access transaction. The bits are only valid when processor access is pending.

<table>
<tr><th>PAA[1:0]</th><th>Psz[1:0]</th><th>Transfer Size</th></tr>
<tr><td>00</td><td>00</td><td>Byte (LE, byte 0; BE, byte 3)</td></tr>
<tr><td>01</td><td>00</td><td>Byte (LE, byte 1; BE, byte 2)</td></tr>
<tr><td>10</td><td>00</td><td>Byte (LE, byte 2; BE, byte 1)</td></tr>
<tr><td>11</td><td>00</td><td>Byte (LE, byte 3; BE, byte 0)</td></tr>
<tr><td>00</td><td>01</td><td>Halfword (LE, bytes 1:0; BE, bytes 3:2)</td></tr>
<tr><td>10</td><td>01</td><td>Halfword (LE, bytes 3:2; BE, bytes 1:0)</td></tr>
<tr><td>00</td><td>10</td><td>Word (LE, BE; bytes 3, 2, 1, 0)</td></tr>
<tr><td>00</td><td>11</td><td>Triple (LE, bytes 2, 1, 0; BE, bytes 3, 2,1)</td></tr>
<tr><td>01</td><td>11</td><td>Triple (LE, bytes 3, 2, 1; BE, bytes 2, 1, 0)</td></tr>
<tr><td colspan="2">All others</td><td>Reserved</td></tr>
</table>

Note: LE=little endian, BE=big endian, the byte# refers to the byte number in a 32-bit register, where byte 3 = bits 31:24; byte 2 = bits 23:16; byte 1 = bits 15:8; byte 0=bits 7:0, independently of the endianess.</td><td>R</td><td>Undefined</td></tr>
<tr><td>Res</td><td>28:23</td><td>reserved</td><td>R</td><td>0</td></tr>
<tr><td>Doze</td><td>22</td><td>Doze state

The Doze bit indicates any kind of low power mode. The value is sampled in the Capture-DR state of the TAP controller:
0: CPU not in low power mode.
1: CPU is in low power mode

Doze includes the Reduced Power (RP) and WAIT power-reduction modes.</td><td>R</td><td>0</td></tr>
</table>

Table 9-23 *EJTAG Control* Register Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>Halt</td><td>21</td><td>Halt state

The Halt bit indicates if the internal system bus clock is running or stopped. The value is sampled in the Capture-DR state of the TAP controller:

0: Internal system clock is running
1: Internal system clock is stopped</td><td>R</td><td>0</td></tr>
<tr><td>PerRst</td><td>20</td><td>Peripheral Reset

When the bit is set to 1, it is only guaranteed that the peripheral reset has occurred in the system when the read value of this bit is also 1. This is to ensure that the setting from the TCK clock domain gets effect in the CPU clock domain, and in peripherals.

When the bit is written to 0, then the bit must also be read as 0 before it is guaranteed that the indication is cleared in the CPU clock domain also.

This bit controls the EJ_PerRst signal on the core.</td><td>R/W</td><td>0</td></tr>
<tr><td>PRnW</td><td>19</td><td>Processor Access Read and Write

This bit indicates if the pending processor access is for a read or write transaction, and the bit is only valid while PrAcc is set:
0: Read transaction
1: Write transaction</td><td>R</td><td>Undefined</td></tr>
<tr><td>PrAcc</td><td>18</td><td>Processor Access (PA)

Read value of this bit indicates if a Processor Access (PA) to the EJTAG memory is pending:
0: No pending processor access
1: Pending processor access

The probe’s software must clear this bit to 0 to indicate the end of the PA. Write of 1 is ignored.

A pending Processor Access is cleared when Rocc is set, but another PA may occur just after the reset if a debug exception occurs.

Finishing a Processor Access is not accepted while the Rocc bit is set. This is to avoid that a Processor Access occurring after the reset is finished due to indication of a Processor Access that occurred before the reset.

The FASTDATA access can clear this bit.</td><td>R/W0</td><td>0</td></tr>
<tr><td>Res</td><td>17</td><td>reserved</td><td>R</td><td>0</td></tr>
</table>

Table 9-23 *EJTAG Control* Register Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	**Bit(s)**			
PrRst	16	Processor Reset (Implementation dependent behavior) When the bit is set to 1, then it is only guaranteed that this setting has taken effect in the system when the read value of this bit is also 1. This is to ensure that the setting from the *TCK* clock domain gets effect in the CPU clock domain, and in peripherals. When the bit is written to 0, then the bit must also be read as 0 before it is guaranteed that the indication is cleared in the CPU clock domain also. This bit controls the *EJ_PrRst* signal. If the signal is used in the system, then it must be ensured that both the processor and all devices required for a reset are properly reset. Otherwise the system may fail or hang. The bit resets itself, since the EJTAG Control register is reset by hard or soft reset.	R/W	0
ProbEn	15	Probe Enable This bit indicates to the CPU if the EJTAG memory is handled by the probe so processor accesses are answered: 0: The probe does not handle EJTAG memory transactions 1: The probe does handle EJTAG memory transactions It is an error by the software controlling the probe if it sets the ProbTrap bit to 1, but resets the ProbEn to 0. The operation of the processor is UNDEFINED in this case. The ProbEn bit is reflected as a read-only bit in the ProbEn bit, bit 0, in the Debug Control Register (DCR). The read value indicates the effective value in the DCR, due to synchronization issues between *TCK* and CPU clock domains; however, it is ensured that change of the ProbEn prior to setting the EjtagBrk bit will have effect for the debug handler executed due to the debug exception. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not: No EJTAGBOOT indication given: 0 EJTAGBOOT indication given: 1	R/W	0 or 1 from EJTAGBOOT

Table 9-23 *EJTAG Control* Register Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
ProbTrap	14	Probe Trap This bit controls the location of the debug exception vector: 0: In normal memory 0xBFC0.0480 1: In EJTAG memory at 0xFF20.0200 in dmseg Valid setting of the ProbTrap bit depends on the setting of the ProbEn bit, see comment under ProbEn bit. The ProbTrap should not be set to 1, for debug exception vector in EJTAG memory, unless the ProbEn bit is also set to 1 to indicate that the EJTAG memory may be accessed. The read value indicates the effective value to the CPU, due to synchronization issues between *TCK* and CPU clock domains; however, it is ensured that change of the ProbTrap bit prior to setting the EjtagBrk bit will have effect for the EjtagBrk. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not: No EJTAGBOOT indication given: 0 EJTAGBOOT indication given: 1	R/W	0 or 1 from EJTAGBOOT
Res	13	reserved	R	0
EjtagBrk	12	EJTAG Break Setting this bit to 1 causes a debug exception to the processor, unless the CPU was in debug mode or another debug exception occurred. When the debug exception occurs, the processor core clock is restarted if the CPU was in low power mode. This bit is cleared by hardware when the debug exception is taken. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not: No EJTAGBOOT indication given: 0 EJTAGBOOT indication given: 1	R/W1	0 or 1 from EJTAGBOOT
Res	11:4	reserved	R	0
DM	3	Debug Mode This bit indicates the debug or non-debug mode: 0: Processor is in non-debug mode 1: Processor is in debug mode The bit is sampled in the *Capture-DR* state of the TAP controller.	R	0
Res	2:0	reserved	R	0

9.4.3 Processor Access Address Register

The Processor Access Address (*PAA*) register is used to provide the address of the processor access in the dmseg, and the register is only valid when a processor access is pending. The length of the Address register is 32 bits, and this register is selected by shifting in the ADDRESS instruction.

9.4.3.1 Processor Access Data Register

The Processor Access Data (*PAD*) register is used to provide data value to and from a processor access. The length of the Data register is 32 bits, and this register is selected by shifting in the DATA instruction.

The register has the written value for a processor access write due to a CPU store to the dmseg, and the output from this register is only valid when a processor access write is pending. The register is used to provide the data value fora processor access read due to a CPU load or fetch from the dmseg, and the register should only be updated with a new value when a processor access write is pending.

The *PAD* register is 32 bits wide. Data alignment is not used for this register, so the value in the *PAD* register matches data on the internal bus. The undefined bytes for a PA write are undefined, and for a *PAD* read then 0 (zero) must be shifted in for the unused bytes.

The organization of bytes in the *PAD* register depends on the endianess of the core, as shown in Figure 9-4 on page 201. The endian mode for debug/kernel mode is determined by the state of the *SI_Endian* input at power-up.

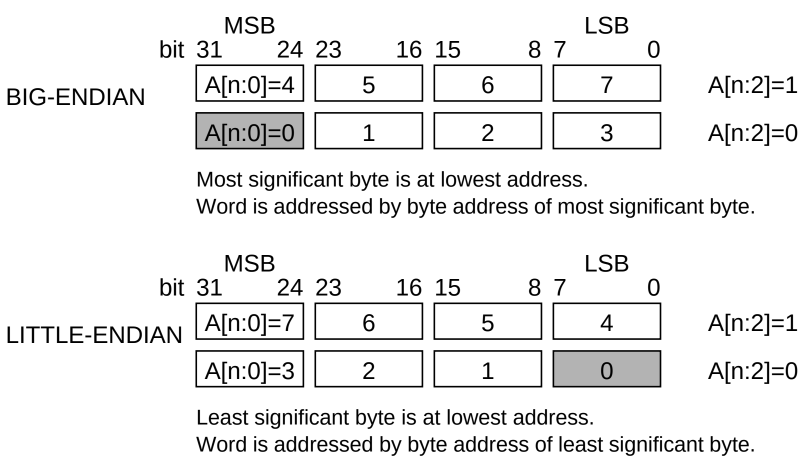


Figure 9-4 Endian Formats for the *PAD* Register

The size of the transaction and thus the number of bytes available/required for the *PAD* register is determined by the Psz field in the *ECR*.

9.4.4 Fastdata Register (TAP Instruction FASTDATA)

The width of the Fastdata register is 1 bit. During a Fastdata access, the Fastdata register is written and read, i.e., a bit is shifted in and a bit is shifted out. During a Fastdata access, the Fastdata register value shifted in specifies whether the Fastdata access should be completed or not. The value shifted out is a flag that indicates whether the Fastdata access was successful or not (if completion was requested).

***Fastdata* Register Format**

0
SPrAcc

Table 9-24 Fastdata Register Field Description

Fields		Description	Read/ Write	Power-up State
Name	Bits			
SPrAcc	0	Shifting in a zero value requests completion of the Fastdata access. The PrAcc bit in the EJTAG Control register is overwritten with zero when the access succeeds. (The access succeeds if PrAcc is one and the operation address is in the legal dmseg Fastdata area.) When successful, a one is shifted out. Shifting out a zero indicates a Fastdata access failure. Shifting in a one does not complete the Fastdata access and the PrAcc bit is unchanged. Shifting out a one indicates that the access would have been successful if allowed to complete and a zero indicates the access would not have successfully completed.	R/W	Undefined

The FASTDATA access is used for efficient block transfers between dmseg (on the probe) and target memory (on the processor). An "upload" is defined as a sequence of processor loads from target memory and stores to dmseg. A "download" is a sequence of processor loads from dmseg and stores to target memory. The "Fastdata area" specifies the legal range of dmseg addresses (0xFF20.0000 - 0xFF20.000F) that can be used for uploads and downloads. The Data + Fastdata registers (selected with the FASTDATA instruction) allow efficient completion of pending Fastdata area accesses.

During Fastdata uploads and downloads, the processor will stall on accesses to the Fastdata area. The PrAcc (processor access pending bit) will be 1 indicating the probe is required to complete the access. Both upload and download accesses are attempted by shifting in a zero SPrAcc value (to request access completion) and shifting out SPrAcc to see if the attempt will be successful (i.e., there was an access pending and a legal Fastdata area address was used). Downloads will also shift in the data to be used to satisfy the load from dmseg's Fastdata area, while uploads will shift out the data being stored to dmseg's Fastdata area.

As noted above, two conditions must be true for the Fastdata access to succeed. These are:

- PrAcc must be 1, i.e., there must be a pending processor access.
- The Fastdata operation must use a valid Fastdata area address in dmseg (0xFF20.0000 to 0xFF20.000F).

Table 9-25 shows the values of the PrAcc and SPrAcc bits and the results of a Fastdata access.

Table 9-25 Operation of the FASTDATA access

Probe Operation	Address Match check	PrAcc in the Control Register	LSB (SPrAcc) shifted in	Action in the Data Register	PrAcc changes to	LSB shifted out	Data shifted out
Download using FASTDATA	Fails	x	x	none	unchanged	0	invalid
	Passes	1	1	none	unchanged	1	invalid
		1	0	write data	0 (SPrAcc)	1	valid (previous) data
		0	x	none	unchanged	0	invalid

Table 9-25 Operation of the FASTDATA access (Continued)

Probe Operation	Address Match check	PrAcc in the Control Register	LSB (SPrAcc) shifted in	Action in the Data Register	PrAcc changes to	LSB shifted out	Data shifted out
Upload using FASTDATA	Fails	x	x	none	unchanged	0	invalid
	Passes	1	1	none	unchanged	1	invalid
		1	0	read data	0 (SPrAcc)	1	valid data
		0	x	none	unchanged	0	invalid

There is no restriction on the contents of the Data register. It is expected that the transfer size is negotiated between the download/upload transfer code and the probe software. Note that the most efficient transfer size is a 32-bit word.

The Rocc bit of the Control register is not used for the FASTDATA operation.

9.5 TAP Processor Accesses

The TAP modules support handling of fetches, loads and stores from the CPU through the dmseg segment, whereby the TAP module can operate like a *slave unit* connected to the on-chip bus. The core can then execute code taken from the EJTAG Probe and it can access data (via a load or store) which is located on the EJTAG Probe. This occurs in a serial way through the EJTAG interface: the core can thus execute instructions e.g. debug monitor code, without occupying the memory.

Accessing the dmseg segment (EJTAG memory) can only occur when the processor accesses an address in the range from 0xFF20.0000 to 0xFF2F.FFFF, the ProbEn bit is set, and the processor is in debug mode (DM=1). In addition the LSNM bit in the CP0 Debug register controls transactions to/from the dmseg.

When a debug exception is taken, while the ProbTrap bit is set, the processor will start fetching instructions from address 0xFF20.0200.

A pending processor access can only finish if the probe writes 0 to PrAcc or by a soft or hard reset.

9.6 Fetch/Load and Store from/to the EJTAG Probe through dmseg

1. The internal hardware latches the requested address into the PA Address register (in case of the Debug exception: 0xFF20.0200).
2. The internal hardware sets the following bits in the EJTAG Control register:
   PrAcc = 1 (selects Processor Access operation)
   PRnW = 0 (selects processor read operation)
   Psz[1:0] = value depending on the transfer size
3. The EJTAG Probe selects the EJTAG Control register, shifts out this control register's data and tests the PrAcc status bit (Processor Access): when the PrAcc bit is found 1, it means that the requested address is available and can be shifted out.
4. The EJTAG Probe checks the PRnW bit to determine the required access.
5. The EJTAG Probe selects the PA Address register and shifts out the requested address.
6. The EJTAG Probe selects the PA Data register and shifts in the instruction corresponding to this address.

7. The EJTAG Probe selects the EJTAG Control register and shifts a PrAcc = 0 bit into this register to indicate to the processor that the instruction is available.
8. The instruction becomes available in the instruction register and the processor starts executing.
9. The processor increments the program counter and outputs an instruction read request for the next instruction. This starts the whole sequence again.

Using the same protocol, the processor can also execute a load instruction to access the EJTAG Probe's memory. For this to happen, the processor must execute a load instruction (e.g. a LW, LH, LB) with the target address in the appropriate range.

Almost the same protocol is used to execute a store instruction to the EJTAG Probe's memory through dmseg. The store address must be in the range: 0xFF20.0000 to 0xFF2F.FFFF, the ProbEn bit must be set and the processor has to be in debug mode (DM=1). The sequence of actions is found below:

1. The internal hardware latches the requested address into the PA Address register
2. The internal hardware latches the data to be written into the PA Data register.
3. The internal hardware sets the following bits in the EJTAG Control register:
PrAcc = 1 (selects Processor Access operation)
PRnW = 1 (selects processor write operation)
Psz[1:0] = value depending on the transfer size
4. The EJTAG Probe selects the EJTAG Control register, shifts out this control register's data and tests the PrAcc status bit (Processor Access): when the PrAcc bit is found 1, it means that the requested address is available and can be shifted out.
5. The EJTAG Probe checks the PRnW bit to determine the required access.
6. The EJTAG Probe selects the PA Address register and shifts out the requested address.
7. The EJTAG Probe selects the PA Data register and shifts out the data to be written.
8. The EJTAG Probe selects the EJTAG Control register and shifts a PrAcc = 0 bit into this register to indicate to the processor that the write access is finished.
9. The EJTAG Probe writes the data to the requested address in its memory.
10. The processor detects that PrAcc bit = 0, which means that it is ready to handle a new access.

The above examples imply that no reset occurs during the operations, and that Rocc is cleared.

9.7 EJTAG Trace

EJTAG Trace enables the ability to trace program flow, load/store addresses and load/store data. Several run-time options exist for the level of information which is traced, including tracing only when in specific processor modes (i.e. UserMode or KernelMode). EJTAG Trace is an optional block in the 4KE core. If EJTAG Trace is not implemented, the rest of this chapter is irrelevant. If EJTAG Trace is implemented, the *CP0 Config3*$_{TL}$ bit is set.

The pipeline specific part of EJTAG Trace is architecturally specified in the *PDtrace™ Interface Specification*. The PDtrace module extracts the trace information from the processor pipeline, and presents it to a pipeline-independent module called the Trace Control Block (TCB). The TCB is specified in the *EJTAG Trace Control Block Specification*. The collective implementation of the two is called *EJTAG Trace*.

When EJTAG Trace is implemented, the 4KE core includes both the PDtrace and the Trace Control Block (TCB) modules. The two modules "talk" to each other on the generic pin-interface called the PDtrace™ Interface. This interface is embedded inside the 4KE core, and will not be discussed in detail here (read the *PDtrace™ Interface Specification*

for a detailed description). While working closely together, the two parts of EJTAG Trace are controlled separately by software. Figure 9-5 shows an overview of the EJTAG Trace modules within the core.

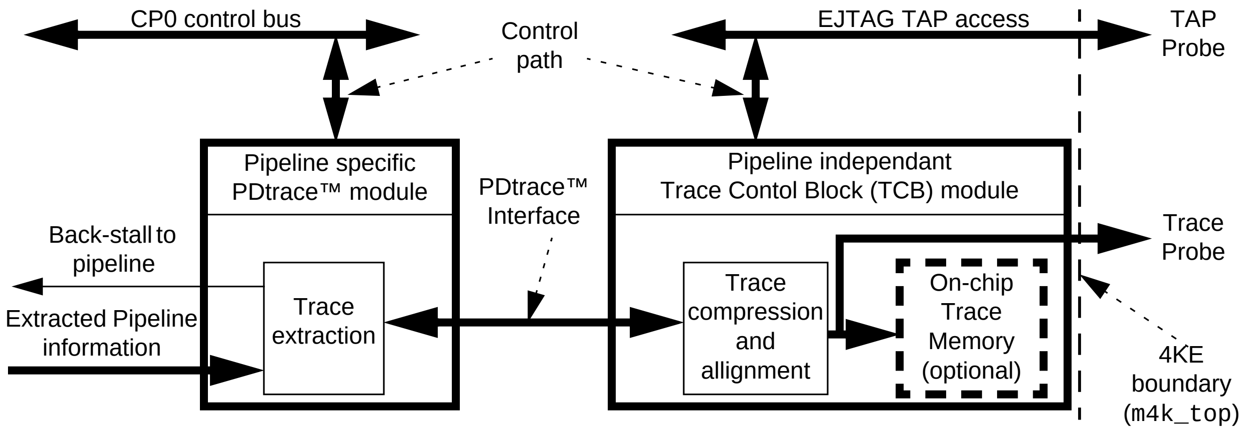


Figure 9-5 EJTAG Trace modules in the 4KE core

To some extent, the two modules both provide similar trace control features, but the access to these features is quite different. The PDtrace controls can only be reached through access to CP0 registers. The TCB controls can only be reached through EJTAG TAP access. The TCB can then control what is traced through the PDtrace™ Interface.

Before describing the EJTAG Trace implemented in the 4KE core, some common terminology and basic features are explained. The remaining sections of this chapter will then provide a more thorough explanation.

9.7.1 Processor Modes

Tracing can be enabled or disabled based on various processor modes. This section precisely describes these modes. The terminology is then used elsewhere in the document.

$$\texttt{DebugMode} \leftarrow (\texttt{Debug}_{DM} = 1)$$
$$\texttt{ExceptionMode} \leftarrow (\texttt{not DebugMode}) \texttt{ and } ((\texttt{Status}_{EXL} = 1) \texttt{ or } (\texttt{Status}_{ERL} = 1))$$
$$\texttt{KernelMode} \leftarrow (\texttt{not (DebugMode or ExceptionMode)}) \texttt{ and } (\texttt{Status}_{UM} = 0)$$
$$\texttt{UserMode} \leftarrow (\texttt{not (DebugMode or ExceptionMode)}) \texttt{ and } (\texttt{Status}_{UM} = 1)$$

9.7.2 Software versus Hardware control

In some of the specifications and in this text, the terms “software control” and “hardware control” are used to refer to the method for how trace is controlled. Software control is when the CP0 register *TraceControl* is used to select the modes to trace, etc. Hardware control is when the EJTAG register *TCBCONTROLA* in the TCB, via the PDtrace interface, is used to select the trace modes. The *TraceControl.TS* bit determines whether software or hardware control is active.

9.7.3 Trace information

The main object of trace is to show the exact program flow from a specific program execution or just a small window of the execution. In EJTAG Trace this is done by providing the minimal cycle-by-cycle information necessary on the PDtrace™ interface for trace regeneration software to reproduce the trace. The following is a summary of the type of information traced:

- Only instructions which complete at the end of the pipeline are traced, and indicated with a completion-flag. The PC is implicitly pointing to the next instruction.

- Load instructions are indicated with a load-flag.
- Store instructions are indicated with a store-flag[1].
- Taken branches are indicated with a branch-taken-flag on the target instruction.
- New PC information for a branch is only traced if the branch target is unpredictable from the static program image.
- When branch targets are unpredictable, only the delta value from current PC is traced, if it is dynamically determined to reduce the number of bits necessary to indicate the new PC. Otherwise the full PC value is traced.
- When a completing instruction is executed in a different processor mode from the previous one, the new processor mode is traced.
- The first instruction is always traced as a branch target, with processor mode and full PC.
- Periodic synchronization instructions are identified with a sync-flag, and traced with the processor mode and full PC.

All the instruction flags above are combined into one 3-bit value, to minimize the bit information to trace. The possible processor modes are explained in Section 9.7.1, "Processor Modes" on page 205.

The target address is statically predictable for all branch and all jump-immediate instructions. If the branch is taken, then the branch-taken-flag will indicate this. All jump-register instructions and ERET/DERET are instructions which have an unpredictable target address. These will have full/delta PC values included in the trace information. Also treated as unpredictable are PC changes which occur due to exceptions, such as an interrupt, reset, etc.

Trace regeneration software is required to know the static program image in memory, in order to reproduce the dynamic flow with the above information. But this is usually not a problem. Only the virtual value of the PC is used. Physical memory location will typically differ.

It is possible to turn on PC delta/full information for all branches, but this should not normally be necessary. As a safety check for trace regeneration software, a periodic synchronization with a full PC is sent. The period of this synchronization is cycle based and programmable.

9.7.4 Load/Store address and data trace information

In addition to PC flow, it is possible to get information on the load/store addresses, as well as the data read/written. When enabled, the following information is optionally added to the trace.

- When load-address tracing is on, the full load address of the first load instruction is traced (indicated by the load-flag). For subsequent loads, a dynamically-determined delta to the previous load address is traced to compress the information which must be sent.
- When store-address tracing is on, the full store address of the first store instruction is traced (indicated by the store-flag). For subsequent stores, a dynamically-determined delta to the previous store address is traced.
- When load-data tracing is on, the full load data read by each load instruction is traced (indicated by the load-flag). Only actual read bytes are traced.
- When store-data tracing is on, the full store data written by each store instruction is traced (indicated by the store-flag). Only written bytes are traced.

After each synchronization instruction, the first load address and the first store address following this are both traced with the full address if load/store address tracing is enabled.

[1] A SC (Store Conditional) instruction is not flagged as a store instruction if the load-locked bit prevented the actual store.

9.7.5 Programmable processor trace mode options

To enable tracing, a global Trace On signal must be set. When trace is on, it is possible to enable tracing in any combination of the processor modes described in Section 9.7.1, "Processor Modes" on page 205. In addition to this, trace can be turned on globally for all process, or only for specific processes by tracing only specific masked values of the ASID found in $EntryHi_{ASID}$ (4KEc cores only).

Additionally, an EJTAG Simple Break trigger point can override the processor mode and ASID selection and turn them all on. Another trigger point can disable this override again.

9.7.6 Programmable trace information options

The processor mode changes are always traced:

- On the first instruction.
- On any synchronization instruction.
- When the mode changes and either the previous or the current processor mode is selected for trace.

The amount of extra information traced is programmable to include:

- PC information only.
- PC and load address.
- PC and store address.
- PC and load and store address.
- PC and load address and load data.
- PC and store address and store data.
- PC and load and store address and load and store data.
- PC and load data only.

The last option is helpful when used together with instruction accurate simulators. If the full internal state of the processor is known prior to trace start, PC and load data are the only information needed to recreate all register values on an instruction by instruction basis.

9.7.6.1 User Data Trace

In addition to the above, a special CP0 register, *UserTraceData*, can generate a data trace. When this register is written, and the global Trace On is set, then the 32-bit data written is put in the trace as special User Data information.

Remark: The User Data is sent even if the processor is operating in an un-traced processor mode.

9.7.7 Enable trace to probe/on-chip memory

When trace is On, based on the options listed in Section 9.7.5, "Programmable processor trace mode options", the trace information is continuously sent on the PDtrace™ interface to the TCB. The TCB must, however, be enabled to transmit the trace information to the Trace probe or to on-chip trace memory, by having the $TCBCONTROLB_{EN}$ bit set. It is possible to enable and disable the TCB in two ways:

- Set/clear the $TCBCONTROLB_{EN}$ bit via an EJTAG TAP operation.
- Initialize a TCB trigger to set/clear the $TCBCONTROLB_{EN}$ bit.

9.7.8 TCB Trigger

The TCB can optionally include 0 to 8 triggers. A TCB trigger can be programmed to fire from any combination of:

- Probe Trigger Input to the TCB.
- Chip-level Trigger Input to the TCB.
- Processor entry into DebugMode.

When a trigger fires it can be programmed to have any combination of actions:

- Create Probe Trigger Output from TCB.
- Create Chip-level Trigger Output from TCB.
- Set, clear, or start countdown to clear the $TCBCONTROLB_{EN}$ bit (start/end/about trigger).
- Put an information byte into the trace stream.

9.7.9 Cycle by cycle information

All of the trace information listed in Section 9.7.3, "Trace information" and Section 9.7.4, "Load/Store address and data trace information", will be collected from the PDtrace™ interface by the TCB. The trace will then be compressed and aligned to fit in 64 bit trace words, with no loss of information. It is possible to exclude/include the exact cycle-by-cycle relationship between each instruction. If excluded, the number of bits required in the trace information from the TCB is reduced, and each trace word will only contain information from completing instructions.

9.7.10 Trace Message Format

The TCB collects trace information every cycle from the PDtrace™ interface. This information is collected into six different Trace Formats (TF1 to TF6). The definition of these Trace Formats is proprietary and will not be released at this time. One important feature is that all Trace Formats have at least one non-zero bit.

9.7.11 Trace Word Format

After the PDtrace™ data has been turned into Trace Formats, the trace information must be streamed to either on-chip trace memory or to the trace probe. Each of the major Trace Formats are of different size. This complicates how to store this information into an on-chip memory of fixed width without too much wasted space. It also complicates how to transmit data through a fixed-width trace probe interface to off-chip memory. To minimize memory overhead and or bandwidth-loss, the Trace Formats are collected into Trace Words of fixed width.

A Trace Word (TW) is defined to be 64 bits wide. An empty/invalid TW is built of all zeros. A TW which contains one or more valid TF's is guaranteed to have a non-zero value on one of the four least significant bits [3:0]. During operation of the TCB, each TW is built from the TF's generated each clock cycle. When all 64 bits are used, the TW is full and can be sent to either on-chip trace memory or to the trace probe. The exact definition of the TW's is proprietary and will not be released at this time.

9.8 PDtrace™ Registers (software control)

The CP0 registers associated with PDtrace are listed in Table 9-26 and described in Chapter 5, "CP0 Registers."

Table 9-26 A List of Coprocessor 0 Trace Registers

Register Number	Sel	Register Name	Reference
23	1	TraceControl	Section 5.2.29, "Trace Control Register (CP0 Register 23, Select 1)" on page 136
23	2	TraceControl2	Section 5.2.30, "Trace Control2 Register (CP0 Register 23, Select 2)" on page 139
23	3	UserTraceData	Section 5.2.31, "User Trace Data Register (CP0 Register 23, Select 3)" on page 141
23	4	TraceBPC	Section 5.2.32, "TraceBPC Register (CP0 Register 23, Select 4)" on page 142

9.9 Trace Control Block (TCB) Registers (hardware control)

The TCB registers used to control its operation are listed in Table 9-27 and Table 9-28. These registers are accessed via the EJTAG TAP interface.

Table 9-27 TCB EJTAG registers

EJTAG Register	Name	Reference	Implemented
0x10	TCBCONTROLA	Section 9.9.1, "TCBCONTROLA Register" on page 209	Yes
0x11	TCBCONTROLB	Section 9.9.2, "TCBCONTROLB Register" on page 212	Yes
0x12	TCBDATA	Section 9.9.3, "TCBDATA Register" on page 216	Yes

Table 9-28 Registers selected by $TCBCONTROLB_{REG}$

$TCBCONTROLB_{REG}$ field	Name	Reference	Implemented
0	TCBCONFIG	Section 9.9.4, "TCBCONFIG Register (Reg 0)" on page 217	Yes
4	TCBTW	Section 9.9.5, "TCBTW Register (Reg 4)" on page 218	Yes if on-chip memory exists. Otherwise No
5	TCBRDP	Section 9.9.6, "TCBRDP Register (Reg 5)" on page 219	
6	TCBWRP	Section 9.9.7, "TCBWRP Register (Reg 6)" on page 219	
7	TCBSTP	Section 9.9.8, "TCBSTP Register (Reg 7)" on page 219	
16-23	TCBTRIGx	Section 9.9.9, "TCBTRIGx Register (Reg 16-23)" on page 220	Only the number indicated by $TCBCONFIG_{TRIG}$ are implemented.

9.9.1 *TCBCONTROLA* Register

The TCB is responsible for asserting or de-asserting the trace input control signals on the PDtrace interface to the core's tracing logic. Most of the control is done using the *TCBCONTROLA* register.

The *TCBCONTROLA* register is written by an EJTAG TAP controller instruction, *TCBCONTROLA* (0x10).

The format of the *TCBCONTROLA* register is shown below, and the fields are described in Table 9-29.

TCBCONTROLA Register Format

31–26	25–24	23	22–20	19	18	17	16	15	14	13	12–5	4	3–1	0
0	VModes	ADW	SyP	TB	IO	D	E	0	K	U	ASID	G	Mode	On

Table 9-29 *TCBCONTROLA* Register Field Descriptions

Fields: Name	Fields: Bits	Description	Read/ Write	Reset State
0	31:26	Reserved. Must be written as zero; returns zero on read.	R	0
VModes	25:24	This field specifies the type of tracing that is supported by the processor, as follows: **Encoding** – **Meaning** 00 – PC tracing only 01 – PC and Load and store address tracing only 10 – PC, load and store address, and load and store data. 11 – Reserved This field is preset to the value of *PDO_ValidModes*.	R	10
ADW	23	*PDO_AD* bus width. 0: The *PDO_AD* bus is 16 bits wide. 1: The *PDO_AD* bus is 32 bits wide.	R	0
SyP	22:20	Used to indicate the synchronization period. The period (in cycles) between which the periodic synchronization information is to be sent is defined as shown in the table below, when the trace buffer is either on-chip or off-chip (as determined by the *TCBCONTROLB*$_{\mathrm{OfC}}$ bit). **SyP** – **On-chip** – **Off-chip** 000 – 2^2 – 2^7 001 – 2^3 – 2^8 010 – 2^4 – 2^9 011 – 2^5 – 2^{10} 100 – 2^6 – 2^{11} 101 – 2^7 – 2^{12} 110 – 2^8 – 2^{13} 111 – 2^9 – 2^{14} This field defines the value on the *PDI_SyncPeriod* signal.	R/W	100
TB	19	Trace All Branches. When set to one, this field indicates that the core must trace either full or incremental PC values for all branches. When set to zero, only the unpredictable branches are traced. This field defines the value on the *PDI_TraceAllBranch* signal.	R/W	Undefined

Table 9-29 *TCBCONTROLA* Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bits			
IO	18	Inhibit Overflow. This bit is used to indicate to the core trace logic that slow but complete tracing is desired. Hence, the core tracing logic must not allow a FIFO overflow and discard trace data. This is achieved by stalling the pipeline when the FIFO is nearly full so that no trace records are ever lost. This field defines the value on the *PDI_InhibitOverflow* signal.	R/W	Undefined
D	17	When set to one, this enables tracing in Debug mode, i.e., when the DM bit is one in the *Debug* register. For trace to be enabled in Debug mode, the On bit must be one and either the G bit must be one, or the current process must match the ASID field in this register. When set to zero, trace is disabled in Debug mode, irrespective of other bits. This field defines the value on the *PDI_DM* signal.	R/W	Undefined
E	16	This controls when tracing is enabled. When set, tracing is enabled when either of the EXL or ERL bits in the *Status* register is one, provided that the On bit (bit 0) is also set, and either the G bit is set, or the current process ASID matches the ASID field in this register. This field defines the value on the *PDI_E* signal.	R/W	Undefined
0	15	Reserved. Must be written as zero; returns zero on read.	R	0
K	14	When set, this enables tracing when the On bit is set and the core is in Kernel mode. Unlike the usual definition of Kernel Mode, this bit enables tracing only when the ERL and EXL bits in the *Status* register are zero. This is provided the On bit (bit 0) is also set, and either the G bit is set, or the current process ASID matches the ASID field in this register. This field defines the value on the *PDI_K* signal.	R/W	Undefined
U	13	When set, this enables tracing when the core is in User mode as defined in the MIPS32 or MIPS64 architecture specification. This is provided the On bit (bit 0) is also set, and either the G bit is set, or the current process ASID matches the ASID field in this register. This field defines the value on the *PDI_U* signal.	R/W	Undefined
ASID	12:5	The ASID field to match when the G bit is zero. When the G bit is one, this field is ignored. On 4KEm and 4KEp cores, this field is ignored. This field defines the value on the *PDI_ASID* signal.	R/W	Undefined
G	4	When set, this implies that tracing is to be enabled for all processes, provided that other enabling functions (like U, S, etc.,) are also true. On 4KEm and 4KEp cores, this field is ignored. This field defines the value on the *PDI_G* signal.	R/W	Undefined

Table 9-29 *TCBCONTROLA* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Mode</td><td>3:1</td><td>When tracing is turned on, this signal specifies what information is to be traced by the core.
<table>
<tr><th>Mode</th><th>Trace Mode</th></tr>
<tr><td>000</td><td>Trace PC</td></tr>
<tr><td>001</td><td>Trace PC and load address</td></tr>
<tr><td>010</td><td>Trace PC and store address</td></tr>
<tr><td>011</td><td>Trace PC and both load/store addresses</td></tr>
<tr><td>100</td><td>Currently un-implemented</td></tr>
<tr><td>101</td><td>Trace PC and load address and data</td></tr>
<tr><td>110</td><td>Trace PC and store address and data</td></tr>
<tr><td>111</td><td>Trace PC and both load/store address and data</td></tr>
</table>The VModes field determines which of these encodings are supported by the processor. The operation of the processor is UNPREDICTABLE if Mode is set to a value which is not supported by the processor

This field defines the value on the PDI_TraceMode signal.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>On</td><td>0</td><td>This is the global trace enable switch to the core. When zero, tracing from the core is always disabled, unless enabled by core internal software override of the PDI_* input pins.

When set to one, tracing is enabled whenever the other enabling functions are also true.

This field defines the value on the PDI_TraceOn signal.</td><td>R/W</td><td>0</td></tr>
</table>

9.9.2 *TCBCONTROLB* Register

The TCB includes a second control register, *TCBCONTROLB* (0x11). This register generally controls what to do with the trace information received.

The format of the *TCBCONTROLB* register is shown below, and the fields are described in Table 9-30.

***TCBCONTROLB* Register Format**

31	30 26	25 21	20	19 17	16	15	14	13 12	11	10 8	7	6 3	2	1	0
WE	0	REG	WR	0	RM	TR	BF	TM	0	CR	Cal	0	CA	OfC	EN

Table 9-30 *TCBCONTROLB* Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>WE</td><td>31</td><td>Write Enable.

Only when set to 1 will the other bits be written in TCBCONTROLB.

This bit will always read 0.</td><td>R</td><td>0</td></tr>
</table>

Table 9-30 *TCBCONTROLB* Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	**Bits**			
0	30:26	Reserved. Must be written as zero; returns zero on read.	R	0
REG	25:21	Register select: This field select the registers accessible through the *TCBDATA* register. Legal values are shown in Table 9-28.	R/W	0
WR	20	Write Registers: When set, the register selected by REG field is read and written when *TCBDATA* is accessed. Otherwise the selected register is only read.	R/W	0
0	19:17	Reserved. Must be written as zero; returns zero on read.	R	0
RM	16	Read on-chip trace memory. When written to 1, the read address-pointer of the on-chip memory is set to point to the oldest memory location written since the last reset of pointers. Subsequent access to the *TCBTW* register (through the *TCBDATA* register), will automatically increment the read pointer (*TCBRDP* register) after each read. [Note: The read pointer does not auto-increment if the WR field is one.] When the write pointer is reached, this bit is automatically reset to 0, and the *TCBTW* register will read all zeros. Once set to 1, writing 1 again will have no effect. The bit is reset by setting the TR bit or by reading the last Trace word in *TCBTW*. This bit is reserved if on-chip memory is not implemented.	R/W1	0
TR	15	Trace memory reset. When written to one, the address pointers for the on-chip trace memory are reset to zero. Also the RM bit is reset to 0. This bit is automatically de-asserted back to 0, when the reset is completed. This bit is reserved if on-chip memory is not implemented.	R/W1	0
BF	14	Buffer Full indicator that the TCB uses to communicate to external software in the situation that the on-chip trace memory is being deployed in the **trace-from** and **trace-to** mode. (See Section 9.13, "TCB On-Chip Trace Memory") This bit is cleared when writing 1 to the TR bit This bit is reserved if on-chip memory is not implemented.	R	0

Table 9-30 *TCBCONTROLB* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>TM</td><td>13:12</td><td>Trace Mode. This field determines how the trace memory is filled when using the simple-break control in the PDtrace™ interface to start or stop trace.
<table><tr><th>TM</th><th>Trace Mode</th></tr><tr><td>00</td><td>Trace-To</td></tr><tr><td>01</td><td>Trace-From</td></tr><tr><td>10</td><td>Reserved</td></tr><tr><td>11</td><td>Reserved</td></tr></table>
In Trace-To mode, the on-chip trace memory is filled, continuously wrapping around and overwriting older Trace Words, as long as there is trace data coming from the core.

In Trace-From mode, the on-chip trace memory is filled from the point that PDO_IamTracing is asserted, and until the on-chip trace memory is full.

In both cases, de-asserting the EN bit in this register will also stop fill to the trace memory.

If a TCBTRIGx trigger control register is used to start/stop tracing, then this field should be set to Trace-To mode.

This bit is reserved if on-chip memory is not implemented.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>11</td><td>Reserved. Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>CR</td><td>10:8</td><td>Off-chip Clock Ratio. Writing this field, sets the ratio of the core clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 9-31 on page 216.

Remark: As the Probe interface works in double data rate (DDR) mode, a 1:2 ratio indicates one data packet sent per core clock rising edge.

This bit is reserved if off-chip trace option is not implemented.</td><td>R/W</td><td>100</td></tr>
</table>

Table 9-30 *TCBCONTROLB* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Cal</td><td>7</td><td>Calibrate off-chip trace interface.

If set to one, the off-chip trace pins will produce the following pattern in consecutive trace clock cycles. If more than 4 data pins exist, the pattern is replicated for each set of 4 pins. The pattern repeats from top to bottom until the Cal bit is de-asserted.

<table>
<tr><th></th><th colspan="4">Calibrations pattern</th></tr>
<tr><th></th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="13">This pattern is replicated for every 4 bits of TR_DATA pins.</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td></tr>
</table>

Note: The clock source of the TCB and PIB must be running.

This bit is reserved if off-chip trace option is not implemented.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>6:3</td><td>Reserved. Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>CA</td><td>2</td><td>Cycle accurate trace.

When set to 1, the trace will include stall information. When set to 0, the trace will exclude stall information, and remove bit zero from all transmitted TF’s.

The stall information included/excluded is:

• TF6 formats with TCBcode 0001 and 0101.

• All TF1 formats.</td><td>R/W</td><td>0</td></tr>
<tr><td>OfC</td><td>1</td><td>If set to 1, trace is sent to off-chip memory using TR_DATA pins.

If set to 0, trace info is sent to on-chip memory.

This bit is read only if a single memory option exists (either off-chip or on-chip only).</td><td>R/W</td><td>Preset</td></tr>
</table>

Table 9-30 *TCBCONTROLB* Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Name	Bits			
EN	0	Enable trace. This is the master enable for trace to be generated from the TCB. This bit can be set or cleared, either by writing this register or from a start/stop/about trigger. When set to 1, trace information is sampled on the *PDO_** pins. Trace Words are generated and sent to either on-chip memory or to the Trace Probe. The target of the trace is selected by the OfC bit. When set to 0, trace information on the *PDO_** pins is ignored. A potential TF6-stop (from a stop trigger) is generated as the last information, the TCB pipe-line is flushed, and trace output is stopped.	R/W	0

Table 9-31 Clock Ratio encoding of the CR field

CR/CRMin/CRMax	Clock Ratio
000	8:1 (Trace clock is eight times that of core clock)
001	4:1 (Trace clock is four times that of core clock)
010	2:1 (Trace clock is double that of core clock)
011	1:1 (Trace clock is same as core clock)
100	1:2 (Trace clock is one half of core clock)
101	1:4 (Trace clock is one fourth of core clock)
110	1:6 (Trace clock is one sixth of core clock)
111	1:8 (Trace clock is one eighth of core clock)

9.9.3 *TCBDATA* Register

The *TCBDATA* register (0x12) is used to access the registers defined by the $TCBCONTROLB_{REG}$ field; see Table 9-28. Regardless of which register or data entry is accessed through *TCBDATA*, the register is only written if the $TCBCONTROLB_{WR}$ bit is set. For read-only registers, the $TCBCONTROLB_{WR}$ is a don’t care.

The format of the *TCBDATA* register is shown below, and the field is described in Table 9-32. The width of *TCBDATA* is 64 bits when on-chip trace words (TWs) are accessed (*TCBTW* access).

***TCBDATA* Register Format**

31(63)	0
Data	

Table 9-32 *TCBDATA* Register Field Descriptions

Fields		Description	Read/Write	Reset State
Names	Bits			
Data	31:0 63:0	Register fields or data as defined by the *TCBCONTROLB*$_{REG}$ field	Only writable if *TCBCONTROLB*$_{WR}$ is set	0

9.9.4 *TCBCONFIG* Register (Reg 0)

The *TCBCONFIG* register holds information about the hardware configuration of the TCB. The format of the *TCBCONFIG* register is shown below, and the field is described in Table 9-33.

***TCBCONFIG* Register Format**

31	30–25	24–21	20–17	16–14	13–11	10–9	8–6	5	4	3–0
CF1	0	TRIG	SZ	CRMax	CRMin	PW	PiN	OnT	OfT	REV

Table 9-33 *TCBCONFIG* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Name	Bits			
CF1	31	This bit is set if a *TCBCONFIG1* register exists. In this revision, *TCBCONFIG1* does not exist and this bit always reads zero.	R	0
0	30:25	Reserved. Must be written as zero; returns zero on read.	R	0
TRIG	24:21	Number of triggers implemented. This also indicates the number of *TCBTRIGx* registers that exist.	R	Legal values are 0 - 8
SZ	20:17	On-chip trace memory size. This field holds the encoded size of the on-chip trace memory. The size in bytes is given by $2^{(SZ+8)}$, implying that the minimum size is 256 bytes and the largest is 8Mb. This bit is reserved if on-chip memory is not implemented.	R	Preset
CRMax	16:14	Off-chip Maximum Clock Ratio. This field indicates the maximum ratio of the core clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 9-31 on page 216. This bit is reserved if off-chip trace option is not implemented.	R	Preset
CRMin	13:11	Off-chip Minimum Clock Ratio. This field indicates the minimum ratio of the core clock to the off-chip trace memory interface clock.The clock-ratio encoding is shown in Table 9-31 on page 216. This bit is reserved if off-chip trace option is not implemented.	R	Preset

Table 9-33 *TCBCONFIG* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>PW</td><td>10:9</td><td>Probe Width: Number of bits available on the off-chip trace interface TR_DATA pins. The number of TR_DATA pins is encoded, as shown in the table.
<table><tr><th>PW</th><th>Number of bits used on TR_DATA</th></tr><tr><td>00</td><td>4 bits</td></tr><tr><td>01</td><td>8 bits</td></tr><tr><td>10</td><td>16 bits</td></tr><tr><td>11</td><td>reserved</td></tr></table>
This field is preset based on input signals to the TCB and the actual capability of the TCB.

This bit is reserved if off-chip trace option is not implemented.</td><td>R</td><td>Preset</td></tr>
<tr><td>PiN</td><td>8:6</td><td>Pipe number.

Indicates the number of execution pipelines.</td><td>R</td><td>0</td></tr>
<tr><td>OnT</td><td>5</td><td>When set, this bit indicates that on-chip trace memory is present. This bit is preset based on the selected option when the TCB is implemented.</td><td>R</td><td>Preset</td></tr>
<tr><td>OfT</td><td>4</td><td>When set, this bit indicates that off-chip trace interface is present. This bit is preset based on the selected option when the TCB is implemented, and on the existence of a PIB module (TC_PibPresent asserted).</td><td>R</td><td>Preset</td></tr>
<tr><td>REV</td><td>3:0</td><td>Revision of TCB. An implementation that conforms to the described architecture in this document must have revision 0.</td><td>R</td><td>0</td></tr>
</table>

9.9.5 *TCBTW* Register (Reg 4)

The *TCBTW* register is used to read Trace Words from the on-chip trace memory. The TW read is the one pointed to by the *TCBRDP* register. A side effect of reading the *TCBTW* register is that the *TCBRDP* register increments to the next TW in the on-chip trace memory. If *TCBRDP* is at the max size of the on-chip trace memory, the increment wraps back to address zero.

This register is reserved if on-chip trace memory is not implemented.

The format of the *TCBTW* register is shown below, and the field is described in Table 9-34.

***TCBTW* Register Format**

63 ... 0
Data

Table 9-34 *TCBTW* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Names	Bits			
Data	63:0	Trace Word	R/W	0

9.9.6 *TCBRDP* Register (Reg 5)

The *TCBRDP* register is the address pointer to on-chip trace memory. It points to the TW read when reading the *TCBTW* register. When writing the $TCBCONTROLB_{RM}$ bit to 1, this pointer is reset to the current value of *TCBSTP*.

This register is reserved if on-chip trace memory is not implemented.

The format of the *TCBRDP* register is shown below, and the field is described in Table 9-35. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, lower three bits are always zero.

***TCBRDP* Register Format**

31 n+1	n 0
	Address

Table 9-35 *TCBRDP* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Names	Bits			
Data	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
Address	n:0	Byte address of on-chip trace memory word.	R/W	0

9.9.7 *TCBWRP* Register (Reg 6)

The *TCBWRP* register is the address pointer to on-chip trace memory. It points to the location where the next new TW for on-chip trace will be written.

This register is reserved if on-chip trace memory is not implemented.

The format of the *TCBWRP* register is shown below, and the fields are described in Table 9-36. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, the lower three bits are always zero.

***TCBWRP* Register Format**

31 n+1	n 0
	Address

Table 9-36 *TCBWRP* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Names	Bits			
Data	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
Address	n:0	Byte address of on-chip trace memory word.	R/W	0

9.9.8 *TCBSTP* Register (Reg 7)

The *TCBSTP* register is the start pointer register. This register points to the on-chip trace memory address at which the oldest TW is located. This pointer is reset to zero when the $TCBCONTROLB_{TR}$ bit is written to 1. If a continuous trace to on-chip memory wraps around the on-chip memory, *TSBSTP* will have the same value as *TCBWRP*.

This register is reserved if on-chip trace memory is not implemented.

The format of the *TCBSTP* register is shown below, and the fields are described in Table 9-37. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, lower three bits are always zero.

***TCBSTP* Register Format**

31 n+1	n 0
	Address

Table 9-37 *TCBSTP* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Names	**Bits**			
Data	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0
Address	n:0	Byte address of on-chip trace memory word.	R/W	0

9.9.9 *TCBTRIGx* Register (Reg 16-23)

Up to eight Trigger Control registers are possible. Each register is named *TCBTRIGx*, where *x* is a single digit number from 0 to 7 (*TCBTRIG0* is Reg 16). The actual number of trigger registers implemented is defined in the *TCBCONFIG*$_{\text{TRIG}}$ field. An unimplemented register will read all zeros and writes are ignored.

Each Trigger Control register controls when an associated trigger is fired, and the action to be taken when the trigger occurs. Please also read Chapter 9, "EJTAG Debug Support," on page 225, for detailed description of trigger logic issues.

The format of the *TCBTRIGx* register is shown below, and the fields are described in Table 9-38.

***TCBTRIGx* Register Format**

31 24	23	22 17	16	15	14 13 7	6	5	4	3 2	1	0
TCBinfo	Trace	0	CH Tro	PD Tro	0	DM	CH Tri	PD Tri	Type	FO	TR

Table 9-38 *TCBTRIGx* Register Field Descriptions

Fields		Description	Read/ Write	Reset State
Names	**Bits**			
TCBinfo	31:24	TCBinfo to be used in a possible TF6 trace format when this trigger fires.	R/W	0
Trace	23	When set, generate TF6 trace information when this trigger fires. Use TCBinfo field for the TCBinfo of TF6 and use Type field for the two MSB of the TCBtype of TF6. The two LSB of TCBtype are 00. The write value of this bit always controls the behavior of this trigger. When this trigger fires, the read value will change to indicate if the TF6 format was ever suppressed by a simultaneous trigger. If so, the read value will be 0. If the write value was 0, the read value is always 0. This special read value is valid until the *TCBTRIGx* register is written.	R/W	0

Table 9-38 *TCBTRIGx* Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State
Names	Bits			
0	22:16	Reserved. Must be written as zero; returns zero on read.	R	0
CHTro	15	When set, generate a single cycle strobe on *TC_ChipTrigOut* when this trigger fires.	R/W	0
PDTro	14	When set, generate a single cycle strobe on *TC_ProbeTrigOut* when this trigger fires.	R/W	0
0	13:7	Reserved. Must be written as zero; returns zero on read.	R	0
DM	6	When set, this Trigger will fire when a rising edge on the Debug mode indication from the core is detected. The write value of this bit always controls the behavior of this trigger. When this trigger fires, the read value will change to indicate if this source was ever the cause of a trigger action (even if the action was suppressed). If so the read value will be 1. If the write value was 0 the read value is always 0. This special read value is valid until the *TCBTRIGx* register is written.	R/W	0
CHTri	5	When set, this Trigger will fire when a rising edge on *TC_ChipTrigIn* is detected. The write value of this bit always controls the behavior of this trigger. When this trigger fires, the read value will change to indicate if this source was ever the cause of a trigger action (even if the action was suppressed). If so the read value will be 1. If the write value was 0 the read value is always 0. This special read value is valid until the *TCBTRIGx* register is written.	R/W	0
PDTri	4	When set, this Trigger will fire when a rising edge on *TC_ProbeTrigIn* is detected. The write value of this bit always controls the behavior of this trigger. When this trigger fires, the read value will change to indicate if this source was ever the cause of a trigger action (even if the action was suppressed). If so the read value will be 1. If the write value was 0 the read value is always 0. This special read value is valid until the *TCBTRIGx* register is written.	R/W	0

Table 9-38 *TCBTRIGx* Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Names</th><th>Bits</th></tr>
<tr><td>Type</td><td>3:2</td><td>
Trigger Type: The Type indicates the action to take when this trigger fires. The table below show the Type values and the Trigger action.
<table>
<tr><th>Type</th><th>Trigger action</th></tr>
<tr><td>00</td><td>Trigger Start: Trigger start-point of trace.</td></tr>
<tr><td>01</td><td>Trigger End: Trigger end-point of trace.</td></tr>
<tr><td>10</td><td>Trigger About: Trigger center-point of trace.</td></tr>
<tr><td>11</td><td>Trigger Info: No action trigger, only for trace info.</td></tr>
</table>
The actual action is to set or clear the $TCBCONTROLB_{EN}$ bit. A Start trigger will set $TCBCONTROLB_{EN}$, a End trigger will clear $TCBCONTROLB_{EN}$. The About trigger will clear $TCBCONTROLB_{EN}$ half way through the trace memory, from the trigger. The size determined by the $TCBCONFIG_{SZ}$ field for on-chip memory. Or from the $TCBCONTROLA_{SyP}$ field for off-chip trace.
If Trace is set, then a TF6 format is added to the trace words. For Start and Info triggers this is done before any other TF’s in that same cycle. For End and About triggers, the TF6 format is added after any other TF’s in that same cycle.
If the $TCBCONTROLB_{TM}$ field is implemented it must be set to Trace-To mode (00), for the Type field to control on-chip trace fill.
The write value of this bit always controls the behavior of this trigger.
When this trigger fires, the read value will change to indicate if the trigger action was ever suppressed. If so the read value will be 11. If the write value was 11 the read value is always 11. This special read value is valid until the TCBTRIGx register is written.
</td><td>R/W</td><td>0</td></tr>
<tr><td>FO</td><td>1</td><td>Fire Once. When set, this trigger will not re-fire until the TR bit is de-asserted. When de-asserted this trigger will fire each time one of the trigger sources indicates trigger.</td><td>R/W</td><td>0</td></tr>
<tr><td>TR</td><td>0</td><td>
Trigger happened. When set, this trigger fired since the TR bit was last written 0.
This bit is used to inspect whether the trigger fired since this bit was last written zero.
When set, all the trigger source bits (bit 4 to 13) will change their read value to indicate if the particular bit was the source to fire this trigger. Only enabled trigger sources can set the read value, but more than one is possible.
Also when set the Type field and the Trace field will have read values which indicate if the trigger action was ever suppressed by a higher priority trigger.
</td><td>R/W0</td><td>0</td></tr>
</table>

9.9.10 Register Reset State

Reset state for all register fields is entered when either of the following occur:

1. TAP controller enters/is in Test-Logic-Reset state.
2. *EJ_TRST_N* input is asserted low.

9.10 EJTAG Trace Enabling

As there are several ways to enable tracing, it can be quite confusing to figure out how to turn tracing on and off. This section should help clarify the enabling of trace.

9.10.1 Trace Trigger from EJTAG Hardware Instruction/Data Breakpoints

If hardware instruction/data simple breakpoints are implemented in the 4KE core, then these breakpoint can be used as triggers to start/stop trace. When used for this, the breakpoints need not also generate a debug exception, but are capable of only generating an internal trigger to the trace logic. This is done by only setting the TE bit and not the BE bit in the Breakpoint Control register. Please see Section 9.2.8.5, "Instruction Breakpoint Control n (IBCn) Register" on page 178 and Section 9.2.9.5, "Data Breakpoint Control n (DBCn) Register" on page 184, for details on breakpoint control.

In connection with the breakpoints, the Trace BreakPoint Control (*TraceBPC*) register is used to define the trace action when a trigger happens. When a breakpoint is enabled as a trigger (TE = 1), it can be selected to be either a start or a stop trigger to the trace logic. Please see Section 5.2.32, "TraceBPC Register (CP0 Register 23, Select 4)" on page 142 for detail in how to define a start/stop trigger.

9.10.2 Turning On PDtrace™ Trace

Trace enabling and disabling from software is similar to the hardware method, with the exception that the bits in the control register are used instead of the input enable signals from the TCB. The $TraceControl_{TS}$ bit controls whether hardware (via the TCB), or software (via the *TraceControl* register) controls tracing functionality.

Trace is turned on when the following expression evaluates true:

```
(
   (
      (TraceControl_TS and TraceControl_On) or
      ((not TraceControl_TS) and TCBCONTROLA_On)
   )
   and
   (MatchEnable or TriggerEnable)
)
```

where,

```
MatchEnable ←
(
   TraceControl_TS
   and
   (
      (TraceControl_U and UserMode)      or
      (TraceControl_K and KernelMode)    or
      (TraceControl_E and ExceptionMode) or
      (TraceControl_D and DebugMode)
   )
)
or
(
   (not TraceControl_TS)
   and
   (
      (TCBCONTROLA_U and UserMode)       or
      (TCBCONTROLA_K and KernelMode)     or
      (TCBCONTROLA_E and ExceptionMode)  or
```

```
            (TCBCONTROLA_DM and DebugMode)
        )
    )
```

and where,

```
    TriggerEnable ←
    (
        DBCi_TE          and
        DBS_BS[i]        and
        TraceBPC_DE      and
        (TraceBPC_DBPOn[i] = 1)
    )
    or
    (
        IBCi_TE          and
        IBS_BS[i]        and
        TraceBPC_IE      and
        (TraceBPC_IBPOn[i] = 1)
    )
```

As seen in the expression above, trace can be turned on only if the master switch $\mathit{TraceControl}_{On}$ or $\mathit{TCBCONTROLA}_{On}$ is first asserted.

Once this is asserted, there are two ways to turn on tracing. The first way, the *MatchEnable* expression, uses the input enable signals from the TCB or the bits in the *TraceControl* register. This tracing is done over general program areas. For example, all of the user-level code for a particular process (if ASID is specified), and so on.

The second way to turn on tracing, the *TriggerEnable* expression, is from the processor side using the EJTAG hardware breakpoint triggers. If EJTAG is implemented, and hardware breakpoints can be set, then using this method enables finer grain tracing control. It is possible to send a trigger signal that turns on tracing at a particular instruction. For example, it would be possible to trace a single procedure in a program by triggering on trace at the first instruction, and triggering off trace at the last instruction.

The easiest way to unconditionally turn on trace is to assert either hardware or software tracing and the corresponding trace on signal with other enables. For example, with $\mathit{TraceControl}_{TS}$=0, i.e., hardware controlled tracing, assert $\mathit{TCBCONTROLA}_{On}$, $\mathit{TCBCONTROLA}_{G}$, and all the other signals in the second part of expression *MatchEnable*. To only trace when a particular process with a known ASID is executing, assert $\mathit{TCBCONTROLA}_{On}$, the correct $\mathit{TCBCONTROLA}_{ASID}$ value, and all of $\mathit{TCBCONTROLA}_{U}$, $\mathit{TCBCONTROLA}_{K}$, $\mathit{TCBCONTROLA}_{E}$, and $\mathit{TCBCONTROLA}_{DM}$. (If it is known that the particular process is a user-level process, then it would be sufficient to only assert $\mathit{TCBCONTROLA}_{U}$ for example). When using the EJTAG hardware triggers to turn trace on and off, it is best if $\mathit{TCBCONTROLA}_{On}$ is asserted and all the other processor mode selection bits in *TCBCONTROLA* are turned off. This would be the least confusing way to control tracing with the trigger signals. Tracing can be controlled via software with the *TraceControl* register in a similar manner.

9.10.3 Turning Off PDtrace™ Trace

Trace is turned off when the following expression evaluates true:

```
    (
        (TraceControl_TS and (not TraceControl_On)) and
        ((not TraceControl_TS) and (not TCBCONTROLA_On))
    )
    or
    (
        (not MatchEnable)       and
        (not TriggerEnable)     and
```

```
        TriggerDisable
    )
```

where,

```
    TriggerDisable ←
    (
        DBCi_TE          and
        DBS_BS[i]        and
        TraceBPC_DE      and
        (TraceBPC_DBPOn[i] = 0)
    )
    or
    (
        IBCi_TE          and
        IBS_BS[i]        and
        TraceBPC_IE      and
        (TraceBPC_IBPOn[i] = 0)
    )
```

Tracing can be unconditionally turned off by de-asserting the $TraceControl_{On}$ bit or the $TCBCONTROLA_{On}$ signal. When either of these are asserted, tracing can be turned off if all of the enables are de-asserted, irrespective of the $\mathrm{TraceControl}_{G}$ bit ($TCBCONTROLA_{G}$) and $\mathrm{TraceControl}_{ASID}$ ($TCBCONTROLA_{ASID}$) values. EJTAG hardware breakpoints can be used to trigger trace off as well. Note that if simultaneous triggers are generated, and even one of them turns on tracing, then even if all of the others attempt to trigger trace off, then tracing will still be turned on. This condition is reflected in presence of the "(not TriggerEnable)" term in the expression above.

9.10.4 TCB Trace Enabling

The TCB must be enabled in order to produce a trace on the probe or to on-chip memory, when trace information is sent on the PDtrace™ interface. The main switch for this is the $TCBCONTROLB_{EN}$ bit. When set, the TCB will send trace information to either on-chip trace memory or to the Trace Probe, controlled by the setting of the $TCBCONTROLB_{OfC}$ bit.

The TCB can optionally include trigger logic, which can control the $TCBCONTROLB_{EN}$ bit. Please see Section 9.11, "TCB Trigger logic" for details.

9.10.5 Tracing a reset exception

Tracing a reset exception is possible. However, the $TraceControl_{TS}$ bit is reset to 0 at core reset, so all the trace control must be from the TCB (using *TCBCONTROLA* and *TCBCONTROLB*). The PDtrace fifo and the entire TCB are reset based on an EJTAG reset. It is thus possible to set up the trace modes, etc., using the TAP controller, and then reset the processor core.

9.11 TCB Trigger logic

The TCB is optionally implemented with trigger unit. If this is the case, then the TCBCONFIGTRIG field is non-zero. This section will explain some of the issues around triggers in the TCB.

9.11.1 Trigger units overview

A TCB trigger logic features three main parts.

1. A common Trigger Source detection unit.
2. 1 to 8 separate Trigger Control units.
3. A common Trigger Action unit.

Figure 9-6 show the functional overview of the trigger flow in the TCB.

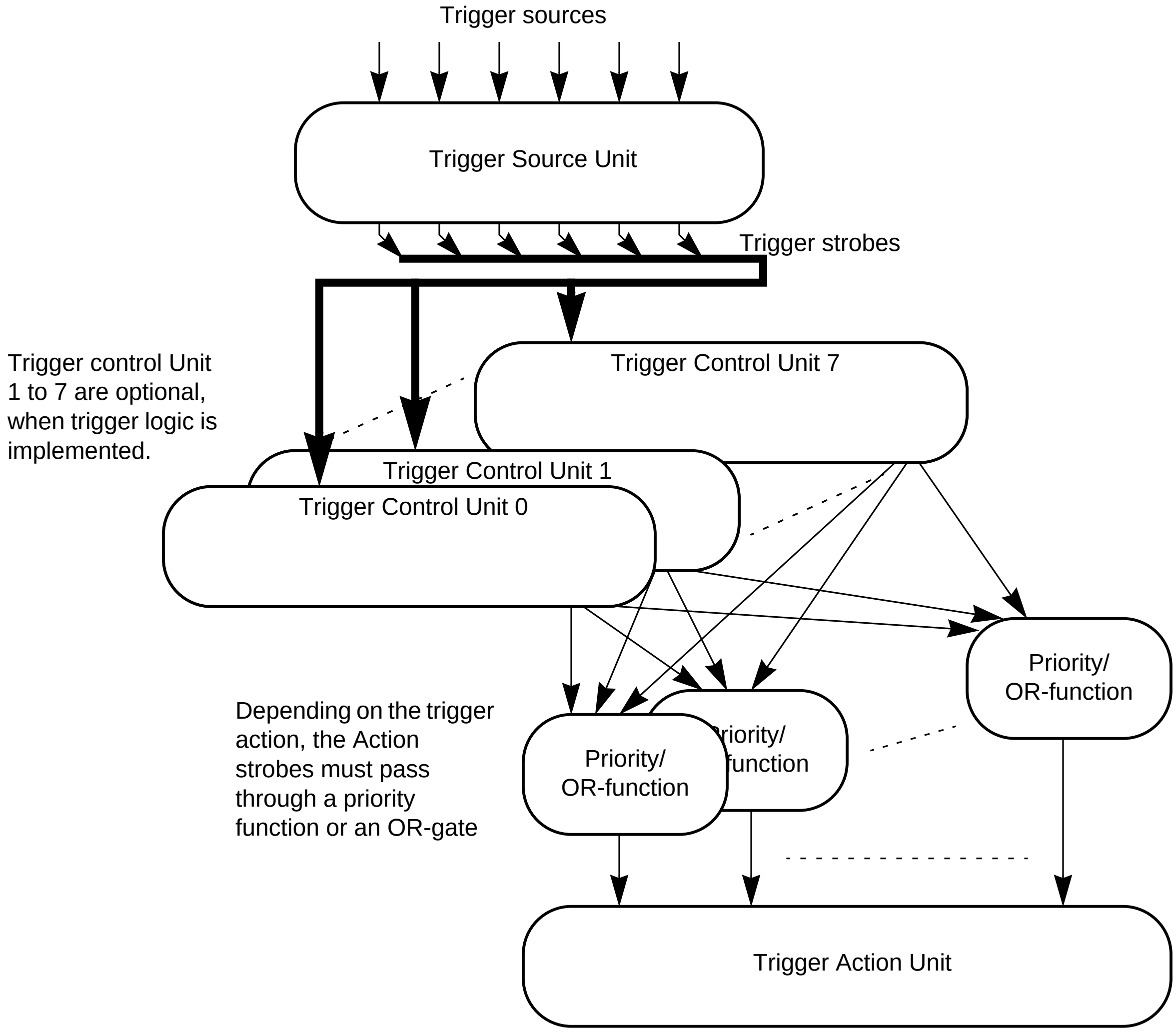


Figure 9-6 TCB Trigger processing overview

9.11.2 Trigger Source Unit

The TCB has three trigger sources:

1.Chip-level trigger input (*TC_ChipTrigIn*).

2. Probe trigger input (*TR_TRIGIN)*.
3. Debug Mode (DM) entry indication from the processor core.

The input triggers are all rising-edge triggers, and the Trigger Source Units convert the edge into a single cycle strobe to the Trigger Control Units.

9.11.3 Trigger Control Units

Up to eight Trigger Control Units are possible. Each of them has it's own Trigger Control Register (*TCBTRIGx*, *x={0..7}*). Each of these registers controls the trigger fire mechanism for the unit. Each unit has all of the Trigger Sources as possible trigger event and they can fire one or more of the Trigger Actions. This is all defined in the Trigger Control register *TCBTRIGx* (see Section 9.9.9, "TCBTRIGx Register (Reg 16-23)" on page 220).

9.11.4 Trigger Action Unit

The TCB has four possible trigger actions:

1. Chip-level trigger output (*TC_ChipTrigOut*).
2. Probe trigger output (*TR_TRIGOUT*).
3. Trace information. Put a programmable byte into the trace stream from the TCB.
4. Start, End or About (delayed end) control of the $TCBCONTROLB_{EN}$ bit.

The basic function of the trigger actions is explained in Section 9.9.9, "TCBTRIGx Register (Reg 16-23)" on page 220. Please also read the next Section 9.11.5, "Simultaneous triggers".

9.11.5 Simultaneous triggers

Two or more triggers can fire simultaneously. The resulting behavior depends on trigger action set for each of them, and whether they should produce a TF6 trace information output or not. There are two groups of trigger actions: Prioritized and OR'ed.

9.11.5.1 Prioritized trigger actions

For prioritized simultaneous trigger actions, the trigger control unit which has the lowest number takes precedence over the higher numbered units. The *x* in *TCBTRIGx* registers defines the number. The oldest trigger takes precedence over everything.

The following trigger actions are prioritized when two or more units fire simultaneously:

- Trigger Start, End and About type triggers ($TCBTRIGx_{Type}$ field set to 00, 01 or 10), which will assert/de-assert the $TCBCONTROLB_{EN}$ bit. The About trigger is delayed and will always change $TCBCONTROLB_{EN}$ because it is the oldest trigger when it de-asserts $TCBCONTROLB_{EN}$. An About trigger will not start the countdown if an even older About trigger is using the Trace Word counter.
- Triggers which produce TF6 trace information in the trace flow (Trace bit is set).

Regardless of priority, the $TCBTRIGx_{TR}$ bit is set when the trigger fires. This is so even if a trigger action is suppressed by a higher priority trigger action. If the trigger is set to only fire once (the $TCBTRIGx_{FO}$ bit is set), then the suppressed trigger action will not happen until after $TCBTRIGx_{TR}$ is written 0.

If a Trigger action is suppressed by a higher priority trigger, then the read value, when the $TCBTRIGx_{TR}$ bit is set, for the $TCBTRIGx_{Trace}$ field will be 0 for suppressed TF6 trace information actions. The read value in the $TCBTRIGx_{Type}$ field for suppressed Start/End/About triggers will be 11. This indication of a suppressed action is sticky. If any of the two actions (Trace and Type) are ever suppressed for a multi-fire trigger (the $TCBTRIGx_{FO}$ bit is zero), then the read values in Trace and/or Type are set to indicate any suppressed action.

About trigger

The About triggers delayed de-assertion of the *TCBCONTROLB*$_{EN}$ bit is always executed, regardless of priority from another Start trigger at the time of the *TCBCONTROLB*$_{EN}$ change. This means that if a simultaneous About trigger action on the *TCBCONTROLB*$_{EN}$ bit (n/2 Trace Words after the trigger) and a Start trigger hit the same cycle, then the About trigger wins, regardless of which trigger number it is. The oldest trigger takes precedence.

However, if an About trigger has started the count down from n/2, but not yet reached zero, then a new About trigger, will NOT be executed. Only one About trigger can have the cycle counter. This second About trigger will store 11 in the *TCBTRIGx*$_{Type}$ field. But, if the *TCBTRIGx*$_{Trace}$ bit is set, a TF6 trace information will still go in the trace.

9.11.5.2 OR'ed trigger actions

The simple trigger actions CHTro and PDTro from each trigger unit, are effectively OR'ed together to produce the final trigger. One or more expected trigger strobes on i.e. *TC_ChipTrigOut* can thus disappear. External logic should not rely on counting of strobes, to predict a specific event, unless simultaneous triggers are known not to occur.

9.12 EJTAG Trace cycle-by-cycle behavior

A key reason for using trace, and not single stepping to debug a software problem, is often to get a picture of the real-time behavior. However the trace logic itself can, when enabled, affect the exact cycle-by-cycle behavior,

9.12.1 Fifo logic in PDtrace and TCB modules

Both the PDtrace module and the TCB module contain a fifo. This might seem like extra overhead, but there are good reasons for this. The vast majority of the information compression happens in the PDtrace module. Any data information, like PC and load/store address values (delta or full), load/store data and processor mode changes, are all sent on the same 16 data bus to the TCB on the PDtrace™ interface. When an instruction requires more than 16 bits of information to be traced properly, the PDtrace fifo will buffer the information, and send it on subsequent clock cycles.

In the TCB, the on-chip trace memory is defined as a 64-bit wide synchronous memory running at core-clock speed. In this case the fifo is not needed. For off-chip trace through the Trace Probe, the fifo comes into play, because only a limited number of pins (4, 8 or 16) exist. Also the speed of the Trace Probe interface can be different (either faster or slower) from that of the 4KE core. So for off-chip tracing, a specific TCB TW fifo is needed.

9.12.2 Handling of Fifo overflow in the PDtrace module

Depending on the amount of trace information selected for trace, and the frequency with which the 16-bit data interface is needed, it is possible for the PDtrace fifo overflow from time to time. There are two ways to handle this case:

1. Allow the overflow to happen, and thereby lose some information from the trace data.
2. Prevent the overflow by back-stalling the core, until the fifo has enough empty slots to accept new trace data.

The PDtrace fifo option is controlled by either the *TraceControl*$_{IO}$ or the *TCBCONTROLA*$_{IO}$ bit, depending on the setting of *TraceControl*$_{TS}$ bit.

The first option is free of any cycle-by-cycle change whether trace is turned on or not. This is achieved at the cost of potentially losing trace information. After an overflow, the fifo is completely emptied, and the next instruction is traced as if it was the start of the trace (processor mode and full PC are traced). This guarantees that only the un-traced fifo information is lost.

The second option guarantees that all the trace information is traced to the TCB. In some cases this is then achieved by back-stalling the core pipeline, giving the PDtrace fifo time to empty enough room in the fifo to accept new trace information from a new instruction. This option can obviously change the real-time behavior of the core when tracing is turned on.

If PC trace information is the only thing enabled (in $TraceControl_{MODE}$ or $TCBCONTROLA_{MODE}$, depending on the setting of $TraceControl_{TS}$), and Trace of all branches is turned off (via $TraceControl_{TB}$ or $TCBCONTROLA_{TB}$, depending on the setting of $TraceControl_{TS}$), then the fifo is unlikely to overflow very often, if at all. This is of course very dependent on the code executed, and the frequency of exception handler jumps, but with this setting there is very little information overhead.

9.12.3 Handling of Fifo overflow in the TCB

The TCB also holds a fifo, used to buffer the TW's which are sent off-chip through the Trace Probe. The data width of the probe can be either 4, 8 or 16 pins, and the speed of these data pins can be from 16 times the core-clock to 1/4 of the core clock (the trace probe clock always runs at a double data rate multiple to the core-clock). See Section 9.12.3.1, "Probe width and Clock-ratio settings" for a description of probe width and clock-ratio options. The combination between the probe width (4, 8 or 16) and the data speed, allows for data rates through the trace probe from 256 bits per core-clock cycle down to only 1 bit per core-clock cycle. The high extreme is not likely to be supported in any implementation, but the low one might be.

The data rate is an important figure when the likelihood of a TCB fifo overflow is considered. The TCB will at maximum produce one full 64-bit TW per core-clock cycle. This is true for any selection of trace mode in $TraceControl_{MODE}$ or $TCBCONTROLA_{MODE}$. The PDtrace module will guarantee the limited amount of data. If the TCB data rate cannot be matched by the off-chip probe width and data speed, then the TCB fifo can possibly overflow. There is only one way to handle this:

1. Prevent the overflow by asserting a stall-signal back to the core (*PDI_StallSending*). This will in turn stall the core pipeline.

There is no way to guarantee that this back-stall from the TCB is never asserted, unless the effective data rate of the Trace Probe interface is at least 64-bits per core-clock cycle.

As a practical matter, the amount of data to the TCB can be minimized by only tracing PC information and excluding any cycle accurate information. This is explained in Section 9.12.2, "Handling of Fifo overflow in the PDtrace module" and below in Section 9.12.4, "Adding cycle accurate information to the trace". With this setting, a data rate of 8-bits per core-clock cycle is usually sufficient. No guarantees can be given here, however, as heavy interrupt activity can increase the number of unpredictable jumps considerably.

9.12.3.1 Probe width and Clock-ratio settings

The actual number of data pins (4, 8 or 16) is defined by the $TCBCONFIG_{PW}$ field. Furthermore, the frequency of the Trace Probe can be different from the core-clock frequency. The trace clock (*TR_CLK*) is a double data rate clock. This means that the data pins (*TR_DATA*) change their value on both edges of the trace clock. When the trace clock is running at clock ratio of 1:2 (one half) of core clock, the data output registers are running a core-clock frequency. The clock ratio is set in the $TCBCONTROLB_{CR}$ field. The legal range for the clock ratio is defined in $TCBCONFIG_{CRMax}$ and $TCBCONFIG_{CRMin}$ (both values inclusive). If $TCBCONTROLB_{CR}$ is set to an unsupported value, the result is UNPREDICABLE. The maximum possible value for $TCBCONFIG_{CRMax}$ is 8:1 (*TR_CLK* is running 8 times faster than core-clock). The minimum possible value for $TCBCONFIG_{CRMin}$ is 1:8 (*TR_CLK* is running at one eighth of the core-clock). See Table 9-31 on page 216 for a description of the encoding of the clock ratio fields.

9.12.4 Adding cycle accurate information to the trace

Depending on the trace regeneration software, it is possible to obtain the exact cycle time relationship between each instruction in the trace. This information is added to the trace, when the $TCBCONTROLB_{CA}$ bit is set. The overhead on the trace information is a little more than one extra bit per core-clock cycle.

This setting only affects the TCB module and not the PDtrace module. The extra bit therefore only affects the likelihood of the TCB fifo overflowing.

9.13 TCB On-Chip Trace Memory

When on-chip trace memory is available ($TCBCONFIG_{OnT}$ is set) the memory is typically of smaller size than if it were external in a trace probe. The assumption is that it is of some value to trace a smaller piece of the program.

With on-chip trace memory, the TCB can work in three possible modes:

1. Trace-From mode.
2. Trace-To mode.
3. Under Trigger unit control.

Software can select this mode using the $TCBCONTROLB_{TM}$ field. If one or more trigger control registers (*TCBTRIGx*) are implemented, and they are using Start, End or About triggers, then the trace mode in $TCBCONTROLB_{TM}$ should be set to Trace-To mode.

9.13.1 On-Chip Trace Memory size

The supported On-chip trace memory size can range from 256 byte to 8Mbytes, in powers of 2. The actual size is shown in the $TCBCONFIG_{SZ}$ field.

9.13.2 Trace-From Mode

In the Trace-From mode, tracing begins when the processor enters into a processor mode/ASID value which is defined to be traced or when an EJTAG hardware breakpoint trace trigger turns on tracing. Trace collection is stopped when the buffer is full. The TCB then signals buffer full using $TCBCONTROLB_{BF}$. When external software polling this register finds the $TCBCONTROLB_{BF}$ bit set, it can then read out the internal trace memory. Saving the trace into the internal buffer will re-commence again only when the $TCBCONTROLB_{BF}$ bit is reset and if the core is sending valid trace data (i.e., *PDO_IamTracing* not equal 0).

9.13.3 Trace-To Mode

In the Trace-To mode, the TCB keeps writing into the internal trace memory, wrapping over and overwriting the oldest information, until the processor is reaches an end of trace condition. End of trace is reached by leaving the processor mode/ASID value which is traced, or when an EJTAG hardware breakpoint trace trigger turns tracing off. At this point, the on-chip trace buffer is then dumped out in a manner similar to that described above in Section 9.13.2, "Trace-From Mode".

Instruction Set Overview

This chapter provides a general overview on the three CPU instruction set formats of the MIPS architecture: Immediate, Jump, and Register. Refer to Chapter 11, “MIPS32™ 4KE™ Processor Core Instructions,” for a complete listing and description of instructions.

This chapter discusses the following topics

10.1 CPU Instruction Formats

Each CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats immediate (I-type), jump (J-type), and register (R-type)—as shown in Figure 10-1 on page 232. The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats as needed.

I-Type (Immediate)

31 26	25 21	20 16	15 0
op	rs	rt	immediate

J-Type (Jump)

31 26	25 0
op	target

R-Type (Register)

31 26	25 21	20 16	15 11	10 6	5 0
op	rs	rt	rd	sa	funct

op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (source/destination) register or branch condition
immediate	16-bit immediate value, branch displacement or address displacement
target	26-bit jump target address
rd	5-bit destination register specifier
sa	5-bit shift amount
funct	6-bit function field

Figure 10-1 Instruction Formats

10.2 Load and Store Instructions

Load and store instructions are immediate (I-type) instructions that move data between memory and the general registers. The only addressing mode that load and store instructions directly support is *base register plus 16-bit signed immediate offset*.

10.2.1 Scheduling a Load Delay Slot

A load instruction that does not allow its result to be used by the instruction immediately following is called a *delayed load instruction*. The instruction slot immediately following this delayed load instruction is referred to as the *load delay slot*.

In a 4KE core, the instruction immediately following a load instruction can use the contents of the loaded register; however in such cases hardware interlocks insert additional real cycles. Although not required, the scheduling of load delay slots can be desirable, both for performance and R-Series processor compatibility.

10.2.2 Defining Access Types

Access type indicates the size of a core data item to be loaded or stored, set by the load or store instruction opcode.

Regardless of access type or byte ordering (endianness), the address given specifies the low-order byte in the addressed field. For a big-endian configuration, the low-order byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the three low-order bits of the address, define the bytes accessed within the addressed word as shown in Table 10-1. Only the combinations shown in Table 10-1 are permissible; other combinations cause address error exceptions.

Table 10-1 Byte Access Within a Word

				Bytes Accessed							
	Low Order Address Bits			Big Endian (31---------------------0)				Little Endian (31---------------------0)			
Access Type	**2**	**1**	**0**	**Byte**				**Byte**			
Word	0	0	0	0	1	2	3	3	2	1	0
Triplebyte	0	0	0	0	1	2			2	1	0
	0	0	1		1	2	3	3	2	1	
Halfword	0	0	0	0	1					1	0
	0	1	0			2	3	3	2		
Byte	0	0	0	0							0
	0	0	1		1					1	
	0	1	0			2			2		
	0	1	1				3	3			

10.3 Computational Instructions

Computational instructions can be either in register (R-type) format, in which both operands are registers, or in immediate (I-type) format, in which one operand is a 16-bit immediate.

Computational instructions perform the following operations on register values:

– Arithmetic

– Logical

– Shift

– Multiply

– Divide

These operations fit in the following four categories of computational instructions:

– ALU Immediate instructions

– Three-operand Register-type Instructions

– Shift Instructions

– Multiply And Divide Instructions

10.3.1 Cycle Timing for Multiply and Divide Instructions

Any multiply instruction in the integer pipeline is transferred to the multiplier as remaining instructions continue through the pipeline; the product of the multiply instruction is saved in the HI and LO registers. If the multiply instruction is followed by an MFHI or MFLO before the product is available, the pipeline interlocks until this product does become available. Refer to Chapter 2, “Pipeline,” on page 11 for more information on instruction latency and repeat rates.

10.4 Jump and Branch Instructions

Jump and branch instructions change the control flow of a program. All jump and branch instructions occur with a delay of one instruction: that is, the instruction immediately following the jump or branch (this is known as the instruction in the *delay slot*) always executes while the target instruction is being fetched from storage.

10.4.1 Overview of Jump Instructions

Subroutine calls in high-level languages are usually implemented with Jump or Jump and Link instructions, both of which are J-type instructions. In J-type format, the 26-bit target address shifts left 2 bits and combines with the high-order 4 bits of the current program counter to form an absolute address.

Returns, dispatches, and large cross-page jumps are usually implemented with the Jump Register or Jump and Link Register instructions. Both are R-type instructions that take the 32-bit byte address contained in one of the general purpose registers.

For more information about jump instructions, refer to the individual instructions in Section 11.3, "MIPS32 Instruction Set for the 4KE core" on page 240.

10.4.2 Overview of Branch Instructions

All branch instruction target addresses are computed by adding the address of the instruction in the delay slot to the 16-bit *offset* (shifted left 2 bits and sign-extended to 32 bits). All branches occur with a delay of one instruction.

If a conditional branch likely is not taken, the instruction in the delay slot is nullified.

Branches, jumps, ERET, and DERET instructions should not be placed in the delay slot of a branch or jump.

10.5 Control Instructions

Control instructions allow the software to initiate traps; they are always R-type.

10.6 Coprocessor Instructions

CP0 instructions perform operations on the System Control Coprocessor registers to manipulate the memory management and exception handling facilities of the processor. Refer to Chapter 11, “MIPS32™ 4KE™ Processor Core Instructions,” on page 237 for a listing of CP0 instructions.

10.7 Enhancements to the MIPS Architecture

The core execution unit implements the MIPS32 architecture, which includes the following instructions.

- CLOCount Leading Ones
- CLZCount Leading Zeros
- MADDMultiply and Add Word
- MADDUMultiply and Add Unsigned Word
- MSUBMultiply and Subtract Word
- MSUBUMultiply and Subtract Unsigned Word
- MULMultiply Word to Register
- SSNOPSuperscalar Inhibit NOP

10.7.1 CLO - Count Leading Ones

The CLO instruction counts the number of leading ones in a word. The 32-bit word in the GPR *rs* is scanned from most-significant to least-significant bit. The number of leading ones is counted and the result is written to the GPR *rd*. If all 32 bits are set in the GPR *rs*, the result written to the GPR *rd* is 32.

10.7.2 CLZ - Count Leading Zeros

The CLZ instruction counts the number of leading zeros in a word. The 32-bit word in the GPR *rs* is scanned from most-significant to least-significant bit. The number of leading zeros is counted and the result is written to the GPR *rd*. If all 32 bits are cleared in the GPR *rs*, the result written to the GPR *rd* is 32.

10.7.3 MADD - Multiply and Add Word

The MADD instruction multiplies two words and adds the result to the HI/LO register pair. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is added to the 64-bit concatenated values in the HI and LO register pair. The resulting value is then written back to the HI and LO registers. No arithmetic exception occurs under any circumstances.

10.7.4 MADDU - Multiply and Add Unsigned Word

The MADDU instruction multiplies two unsigned words and adds the result to the HI/LO register pair. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is added to the 64-bit concatenated values in the HI and LO register pair. The resulting value is then written back to the HI and LO registers. No arithmetic exception occurs under any conditions.

10.7.5 MSUB - Multiply and Subtract Word

The MSUB instruction multiplies two words and subtracts the result from the HI/LO register pair. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values in the HI and LO register pair. The resulting value is then written back to the HI and LO registers. No arithmetic exception occurs under any circumstances.

10.7.6 MSUBU - Multiply and Subtract Unsigned Word

The MSUBU instruction multiplies two unsigned words and subtracts the result from the HI/LO register pair. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values in the HI and LO register pair. The resulting value is then written back to the HI and LO registers. No arithmetic exception occurs under any circumstances.

10.7.7 MUL - Multiply Word

The MUL instruction multiplies two words and writes the result to a GPR. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The least-significant 32-bits of the product are written to the GPR *rd*. The contents of the HI and LO register pair are not defined after the operation. No arithmetic exception occurs under any circumstances.

10.7.8 SSNOP- Superscalar Inhibit NOP

The MIPS32 4KE processor cores treat this instruction as a regular NOP.

MIPS32™ 4KE™ Processor Core Instructions

This chapter supplements the MIPS32 Architecture Reference Manual by describing instruction behavior that is specific to a MIPS32™ 4KE™ processor core. The chapter is divided into the following sections:

- Section 11.1, "Understanding the Instruction Descriptions" on page 237
- Section 11.2, "MIPS32 4KE Opcode Map" on page 237
- Section 11.3, "MIPS32 Instruction Set for the 4KE core" on page 240

The 4KE processor core also supports the MIPS16 ASE to the MIPS32 architecture. The MIPS16 ASE instruction set is described in Chapter 12, “MIPS16 Application-Specific Extension to the MIPS32 Instruction Set,” on page 273.

11.1 Understanding the Instruction Descriptions

Refer to Volume II of the MIPS32 Architecture Reference Manual for more information about the instruction descriptions. There is a description of the instruction fields, definition of terms, and a description function notation available in that document.

11.2 MIPS32 4KE Opcode Map

Key

- CAPITALIZED text indicates an opcode mnemonic
- *Italicized* text indicates to look at the specified opcode submap for further instruction bit decode
- Entries containing the α symbol indicate that a reserved instruction fault occurs if the core executes this instruction.
- Entries containing the β symbol indicate that a coprocessor unusable exception occurs if the core executes this instruction

Table 11-1 Encoding of the *Opcode* Field

opcode		bits 28..26							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	*Special*	*RegImm*	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0*	β	*COP2*	β	BEQL	BNEL	BLEZL	BGTZL
3	011	α	α	α	α	*Special2*	JALX	α	*Special3*
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	α
5	101	SB	SH	SWL	SW	α	α	SWR	CACHE
6	110	LL	β	LWC2	PREF	α	β	α	α
7	111	SC	β	SWC2	α	α	β	α	α

Table 11-2 *Special* Opcode encoding of Function Field

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL	β	SRL/ ROTR	SRA	SLLV	α	SRLV/ ROTRV	SRAV
1	001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	α	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	α	α	α	α
3	011	MULT	MULTU	DIV	DIVU	α	α	α	α
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	α	α	SLT	SLTU	α	α	α	α
6	110	TGE	TGEU	TLT	TLTU	TEQ	α	TNE	α
7	111	α	α	α	α	α	α	α	α

Table 11-3 *Special2* Opcode Encoding of Function Field

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL	α	MSUB	MSUBU	α	α
1	001	α	α	α	α	α	α	α	α
2	010	*UDI*[1] or α							
3	011								
4	100	CLZ	CLO	α	α	α	α	α	α
5	101	α	α	α	α	α	α	α	α
6	110	α	α	α	α	α	α	α	α
7	111	α	α	α	α	α	α	α	SDBBP

1. CorExtend instructions are a build-time option of the 4KE Pro cores, if not implemented this instructions space will cause a reserved instruction exception. If assembler support exists, the mnemonics for CorExtend instructions are most likely UDI0, UDI1, .., UDI15.

Table 11-4 *Special3* Opcode Encoding of Function Field

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	EXT	α	α	α	INS	α	α	α
1	001	α	α	α	α	α	α	α	α
2	010	α	α	α	α	α	α	α	α
3	011	α	α	α	α	α	α	α	α
4	100	BSHFL	α	α	α	α	α	α	α
5	101	α	α	α	α	α	α	α	α
6	110	α	α	α	α	α	α	α	α
7	111	α	α	α	RDHWR	α	α	α	α

Table 11-5 *RegImm* Encoding of rt Field

rt		bits 18..16							
		0	1	2	3	4	5	6	7
bits 20..19		000	001	010	011	100	101	110	111
0	00	BLTZ	BGEZ	BLTZL	BGEZL	α	α	α	α
1	01	TGEI	TGEIU	TLTI	TLTIU	TEQI	α	TNEI	α
2	10	BLTZAL	BGEZAL	BLTZALL	BGEZALL	α	α	α	α
3	11	α	α	α	α	α	α	α	SYNCI

Table 11-6 *COP2* Encoding of rs Field

rs		bits 23..21							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC2	α	CFC2	MFHC2	MTC2	α	CTC2	MTHC2
1	01	BC2	*BC2*[1]						
2	10	CO							
3	11								

1. The core will treat the entire row as a *BC2* instruction. However compiler and assembler support only exists for the first one. Some compiler and assembler products may allow the user to add new instructions.

Table 11-7 *COP2* Encoding of rt Field When rs=*BC2*

rt	bits 16	
bits 17	0	1
0	BC2F	BC2T
1	BC2FL	BC2TL

Table 11-8 *COP0* Encoding of rs Field

rs		bits 23..21							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC0	α	α	α	MTC0	α	α	α
1	01	α	α	RDPGPR	MFMC0	α	α	WRPGPR	α
2	10	CO							
3	11								

Table 11-9 *COP0* Encoding of Function Field When rs=*CO*

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	α	TLBR	TLBWI	α	α	α	TLBWR	α
1	001	TLBP	α	α	α	α	α	α	α
2	010	α	α	α	α	α	α	α	α
3	011	ERET	IACK	α	α	α	α	α	DERET
4	100	WAIT	α	α	α	α	α	α	α
5	101	α	α	α	α	α	α	α	α
6	110	α	α	α	α	α	α	α	α
7	111	α	α	α	α	α	α	α	α

11.3 MIPS32 Instruction Set for the 4KE core

This section describes the MIPS32 instructions for the 4KE cores. Table 11-10 lists the instructions in alphabetical order. Instructions that have implementation dependent behavior are described afterwards. The descriptions for other instructions exist in the architecture reference manual and are not duplicated here.

Table 11-10 Instruction Set

Instruction	Description	Function
ADD	Integer Add	Rd = Rs + Rt
ADDI	Integer Add Immediate	Rt = Rs + Immed
ADDIU	Unsigned Integer Add Immediate	Rt = Rs $+_U$ Immed
ADDU	Unsigned Integer Add	Rd = Rs $+_U$ Rt
AND	Logical AND	Rd = Rs & Rt
ANDI	Logical AND Immediate	Rt = Rs & (0_{16} || Immed)
B	Unconditional Branch (Assembler idiom for: BEQ r0, r0, offset)	PC += (int)offset
BAL	Branch and Link (Assembler idiom for: BGEZAL r0, offset)	GPR[31] = PC + 8 PC += (int)offset
BC2F	Branch On COP2 Condition False	if COP2Condition(cc) == 0 PC += (int)offset
BC2FL	Branch On COP2 Condition False Likely	if COP2Condition(cc) == 0 PC += (int)offset else Ignore Next Instruction
BC2T	Branch On COP2 Condition True	if COP2Condition(cc) == 1 PC += (int)offset
BC2TL	Branch On COP2 Condition True Likely	if COP2Condition(cc) == 1 PC += (int)offset else Ignore Next Instruction
BEQ	Branch On Equal	if Rs == Rt PC += (int)offset

Table 11-10 Instruction Set (Continued)

Instruction	Description	Function
BEQL	Branch On Equal Likely	if Rs == Rt PC += (int)offset else Ignore Next Instruction
BGEZ	Branch on Greater Than or Equal To Zero	if !Rs[31] PC += (int)offset
BGEZAL	Branch on Greater Than or Equal To Zero And Link	GPR[31] = PC + 8 if !Rs[31] PC += (int)offset
BGEZALL	Branch on Greater Than or Equal To Zero And Link Likely	GPR[31] = PC + 8 if !Rs[31] PC += (int)offset else Ignore Next Instruction
BGEZL	Branch on Greater Than or Equal To Zero Likely	if !Rs[31] PC += (int)offset else Ignore Next Instruction
BGTZ	Branch on Greater Than Zero	if !Rs[31] && Rs != 0 PC += (int)offset
BGTZL	Branch on Greater Than Zero Likely	if !Rs[31] && Rs != 0 PC += (int)offset else Ignore Next Instruction
BLEZ	Branch on Less Than or Equal to Zero	if Rs[31] || Rs == 0 PC += (int)offset
BLEZL	Branch on Less Than or Equal to Zero Likely	if Rs[31] || Rs == 0 PC += (int)offset else Ignore Next Instruction
BLTZ	Branch on Less Than Zero	if Rs[31] PC += (int)offset
BLTZAL	Branch on Less Than Zero And Link	GPR[31] = PC + 8 if Rs[31] PC += (int)offset
BLTZALL	Branch on Less Than Zero And Link Likely	GPR[31] = PC + 8 if Rs[31] PC += (int)offset else Ignore Next Instruction
BLTZL	Branch on Less Than Zero Likely	if Rs[31] PC += (int)offset else Ignore Next Instruction
BNE	Branch on Not Equal	if Rs != Rt PC += (int)offset
BNEL	Branch on Not Equal Likely	if Rs != Rt PC += (int)offset else Ignore Next Instruction
BREAK	Breakpoint	Break Exception

Table 11-10 Instruction Set (Continued)

Instruction	Description	Function
CACHE	Cache Operation	See Cache Description
CFC2	Move Control Word From Coprocessor 2	Rt = CCR[2, n]
CLO	Count Leading Ones	Rd = NumLeadingOnes(Rs)
CLZ	Count Leading Zeroes	Rd = NumLeadingZeroes(Rs)
COP0	Coprocessor 0 Operation	See Coprocessor Description
COP2	Coprocessor 2 Operation	See Coprocessor 2 Description
CTC2	Move Control Word To Coprocessor 2	CCR[2, n] = Rt
DERET	Return from Debug Exception	PC = DEPC Exit Debug Mode
DI	Disable Interrupts	Rt=Status $Status_{IE}$=0
DIV	Divide	LO = (int)Rs / (int)Rt HI = (int)Rs % (int)Rt
DIVU	Unsigned Divide	LO = (uns)Rs / (uns)Rt HI = (uns)Rs % (uns)Rt
EHB	Execution Hazard Barrier	Stall until execution hazards are cleared
EI	Enable Interrupts	Rt=Status $Status_{IE}$=1
ERET	Return from Exception	if SR[2] PC = ErrorEPC else PC = EPC SR[1] = 0 SR[2] = 0 LL = 0
EXT	Extract Bit Field	Rt=ExtractField(Rs,msbd,lsb)
IACK	Interrupt Acknowledge	Signal External Interrupt Controller
INS	Insert Bit Field	Rt=InsertField(Rt,Rs,msb,lsb)
J	Unconditional Jump	PC = PC[31:28] || offset<<2
JAL	Jump and Link	GPR[31] = PC + 8 PC = PC[31:28] || offset<<2
JALR	Jump and Link Register	Rd = PC + 8 PC = Rs
JALR.HB	Jump and Link Register with Hazard Barrier	Rd = PC + 8 PC = Rs Stall until all execution and instruction hazards are cleared
JR	Jump Register	PC = Rs

Table 11-10 Instruction Set (Continued)

Instruction	Description	Function
JR.HB	Jump Register with Hazard Barrier	PC = Rs Stall until all execution and instruction hazards are cleared
LB	Load Byte	Rt = (byte)Mem[Rs+offset]
LBU	Unsigned Load Byte	Rt = (ubyte))Mem[Rs+offset]
LH	Load Halfword	Rt = (half)Mem[Rs+offset]
LHU	Unsigned Load Halfword	Rt = (uhalf)Mem[Rs+offset]
LL	Load Linked Word	Rt = Mem[Rs+offset] LL = 1 LLAdr = Rs + offset
LUI	Load Upper Immediate	Rt = immediate << 16
LW	Load Word	Rt = Mem[Rs+offset]
LWC2	Load Word To Coprocessor 2	CPR[2, n, 0] = Mem[Rs+offset]
LWL	Load Word Left	See LWL instruction below.
LWR	Load Word Right	See LWR instruction below.
MADD	Multiply-Add	HI, LO += (int)Rs * (int)Rt
MADDU	Multiply-Add Unsigned	HI, LO += (uns)Rs * (uns)Rt
MFC0	Move From Coprocessor 0	Rt = CPR[0, n, sel]
MFC2	Move From Coprocessor 2	Rt = CPR[2, n, $sel_{31..0}$]
MFHC2	Move From High Word Coprocessor2	Rt= $CPR[2,n,sel]_{63..32}$
MFHI	Move From HI	Rd = HI
MFLO	Move From LO	Rd = LO
MOVN	Move Conditional on Not Zero	if GPR[rt] ≠ 0 then GPR[rd] = GPR[rs]
MOVZ	Move Conditional on Zero	if GPR[rt] = 0 then GPR[rd] = GPR[rs]
MSUB	Multiply-Subtract	HI, LO -= (int)Rs * (int)Rt
MSUBU	Multiply-Subtract Unsigned	HI, LO -= (uns)Rs * (uns)Rt
MTC0	Move To Coprocessor 0	CPR[0, n, sel] = Rt
MTC2	Move To Coprocessor 2	$CPR[2, n, sel]_{31..0}$ = Rt
MTHC2	Move To High Word Coprocessor 2	$CPR[2, n, sel]_{63..32}$ = Rt
MTHI	Move To HI	HI = Rs
MTLO	Move To LO	LO = Rs
MUL	Multiply with register write	HI | LO =Unpredictable Rd = LO
MULT	Integer Multiply	HI | LO = (int)Rs * (int)Rd

Table 11-10 Instruction Set (Continued)

Instruction	Description	Function
MULTU	Unsigned Multiply	HI | LO = (uns)Rs * (uns)Rd
NOP	No Operation (Assembler idiom for: SLL r0, r0, r0)	
NOR	Logical NOR	Rd = ~(Rs | Rt)
OR	Logical OR	Rd = Rs | Rt
ORI	Logical OR Immediate	Rt = Rs | Immed
PREF	Prefetch	Load Specified Line into Cache
RDHWR	Read HardWare Register	Rt=HWR[Rd]
RDPGPR	Read GPR from Previous Shadow Set	Rd=SGPR[SRSCtl$_{PSS}$, Rt]
ROTR	Rotate Word Right	Rd = Rt$_{sa-1..0}$ || Rt$_{31..sa}$
ROTRV	Rotate Word Right Variable	Rd = Rt$_{Rs-1..0}$ || Rt$_{31..Rs}$
SB	Store Byte	(byte)Mem[Rs+offset] = Rt
SC	Store Conditional Word	if LL =1 mem[Rxoffs] = Rt Rt = LL
SDBBP	Software Debug Breakpoint	Trap to SW Debug Handler
SEB	Sign Extend Byte	Rd=SignExtend(Rt$_{7..0}$)
SEH	Sign Extend Half	Rd=SignExtend(Rt$_{15..0}$)
SH	Store Halfword	(half)Mem[Rs+offset] = Rt
SLL	Shift Left Logical	Rd = Rt << sa
SLLV	Shift Left Logical Variable	Rd = Rt << Rs[4:0]

SLT	Set on Less Than	if (int)Rs < (int)Rt
		Rd = 1
		else
		Rd = 0

SLTI	Set on Less Than Immediate	if (int)Rs < (int)Immed
		Rt = 1
		else
		Rt = 0

SLTIU	Set on Less Than Immediate Unsigned	if (uns)Rs < (uns)Immed
		Rt = 1
		else
		Rt = 0

SLTU	Set on Less Than Unsigned	if (uns)Rs < (uns)Immed
		Rd = 1
		else
		Rd = 0

SRA	Shift Right Arithmetic	Rd = (int)Rt >> sa
SRAV	Shift Right Arithmetic Variable	Rd = (int)Rt >> Rs[4:0]
SRL	Shift Right Logical	Rd = (uns)Rt >> sa
SRLV	Shift Right Logical Variable	Rd = (uns)Rt >> Rs[4:0]

Table 11-10 Instruction Set (Continued)

Instruction	Description	Function
SSNOP	Superscalar Inhibit No Operation	Nop
SUB	Integer Subtract	Rt = (int)Rs - (int)Rd
SUBU	Unsigned Subtract	Rt = (uns)Rs - (uns)Rd
SW	Store Word	Mem[Rs+offset] = Rt
SWC2	Store Word From Coprocessor 2	Mem[Rs+offset] = CPR[2, n, 0]
SWL	Store Word Left	See SWL instruction description.
SWR	Store Word Right	See SWR instruction description.
SYNC	Synchronize	See SYNC instruction below.
SYNCI	Synchronize Caches to Make Instruction Writes Effective	Force D$ writeback and I$ invalidate on specified address
SYSCALL	System Call	SystemCallException
TEQ	Trap if Equal	if Rs == Rt TrapException
TEQI	Trap if Equal Immediate	if Rs == (int)Immed TrapException
TGE	Trap if Greater Than or Equal	if (int)Rs >= (int)Rt TrapException
TGEI	Trap if Greater Than or Equal Immediate	if (int)Rs >= (int)Immed TrapException
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	if (uns)Rs >= (uns)Immed TrapException
TGEU	Trap if Greater Than or Equal Unsigned	if (uns)Rs >= (uns)Rt TrapException
TLBP	Probe TLB for Matching Entry	See TLBP instruction below.
TLBR	Read Index for TLB Entry	See TLBR instruction below.
TLBWI	Write Indexed TLB Entry	See TLBWI instruction below.
TLBWR	Write Random TLB Entry	See TLBWR instruction below.
TLT	Trap if Less Than	if (int)Rs < (int)Rt TrapException
TLTI	Trap if Less Than Immediate	if (int)Rs < (int)Immed TrapException
TLTIU	Trap if Less Than Immediate Unsigned	if (uns)Rs < (uns)Immed TrapException
TLTU	Trap if Less Than Unsigned	if (uns)Rs < (uns)Rt TrapException
TNE	Trap if Not Equal	if Rs != Rt TrapException
TNEI	Trap if Not Equal Immediate	if Rs != (int)Immed TrapException
WAIT	Wait for Interrupts	Stall until interrupt occurs

Table 11-10 Instruction Set (Continued)

Instruction	Description	Function
WRPGPR	Write to GPR in Previous Shadow Set	SGPR[SRSCtl$_{PSS}$,Rd]=Rt
WSBH	Word Swap Bytes within Halfwords	Rd=SwapBytesWithinHalfs(Rt)
XOR	Exclusive OR	Rd = Rs ^ Rt
XORI	Exclusive OR Immediate	Rt = Rs ^ (uns)Immed

31–26	25–21	20–16	15–0
CACHE 101111	base	op	offset
6	5	5	16

Format: `CACHE op, offset(base)` **MIPS32**

Purpose:

To perform the cache operation specified by op.

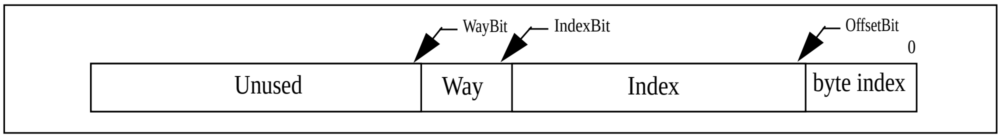
Description:

The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

Table 11-11 Usage of Effective Address

Operation Requires an	Type of Cache	Usage of Effective Address
Address	Physical	The effective address is translated by the MMU to a physical address. The physical address is then used to address the cache
Index	N/A	Assuming that the total cache size in bytes is CS, the associativity is A, and the number of bytes per tag is BPT, the following calculations give the fields of the address which specify the way and the index: OffsetBit ← Log2(BPT) IndexBit ← Log2(CS / A) WayBit ← IndexBit + Ceiling(Log2(A)) Way ← $Addr_{WayBit-1..IndexBit}$ Index ← $Addr_{IndexBit-1..OffsetBit}$ For a direct-mapped cache, the Way calculation is ignored and the Index value fully specifies the cache tag. This is shown symbolically in the figure below.

Figure 11-1 Usage of Address Fields to Select Index and Way



A TLB Refill and TLB Invalid (both with cause code equal TLBL) exception can occur on any operation. For index operations (where the address is used to index the cache but need not match the cache tag) software should use unmapped addresses to avoid TLB exceptions. This instruction never causes TLB Modified exceptions nor TLB Refill exceptions with a cause code of TLBS.

The effective address may be an arbitrarily-aligned by address. The CACHE instruction never causes an Address Error Exception due to an non-aligned address.

A Cache Error exception may occur as a byproduct of some operations performed by this instruction. For example, if a Writeback operation detects a cache or bus error during the processing of the operation, that error is reported via a Cache Error exception. Similarly, a Bus Error Exception may occur if a bus operation invoked by this instruction is terminated in an error. However, cache error exceptions should must be triggered by an Index Load Tag or Index Store tag operation, as these operations are used for initialization and diagnostic purposes.

An address Error Exception (with cause code equal AdEL) occurs if the effective address references a portion of the kernel address space which would normally result in such an exception.Data watch is not triggered by a cache instruction whose address matches the Watch register address match conditions.

Bits [17:16] of the instruction specify the cache on which to perform the operation, as follows:

Table 11-12 Encoding of Bits[17:16] of CACHE Instruction

Code	Name	Cache
2#00	I	Primary Instruction
2#01	D	Primary Data
2#10	T	Not supported
2#11	S	Not supported

Bits [20:18] of the instruction specify the operation to perform.On Index Load Tag and Index Store Data operations, the specific word that is addressed is loaded into / read from the DataLo register. All other cache instructions are line-based and the word and byte indexes will not affect their operation.

Table 11-13 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST,SPR] Cleared

<table>
<tr><th>Code</th><th>Caches</th><th>Name</th><th>Effective Address Operand Type</th><th>Operation</th><th>Implemented?</th></tr>
<tr><td rowspan="3">2#000</td><td>I</td><td>Index Invalidate</td><td>Index</td><td>Set the state of the cache block at the specified index to invalid.

This encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices.</td><td></td></tr>
<tr><td>D</td><td>Index Writeback Invalidate</td><td>Index</td><td rowspan="2">If the state of the cache block at the specified index is valid and dirty, write the block back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid.

This encoding may be used by software to invalidate the entire data cache by stepping through all valid indices. Note that Index Store Tag should be used to initialize the cache at powerup.</td><td>Yes</td></tr>
<tr><td>S, T</td><td>Reserved</td><td>Index</td><td>No</td></tr>
<tr><td>2#001</td><td>I,D</td><td>Index Load Tag</td><td>Index</td><td>Read the tag for the cache block at the specified index into the TagLo Coprocessor 0 register. Also read the data corresponding to the byte index into the DataLo register.</td><td>Yes</td></tr>
<tr><td>2#010</td><td>I,D</td><td>Index Store Tag</td><td>Index</td><td>Write the tag for the cache block at the specified index from the TagLo Coprocessor 0 register.

This encoding may be used by software to initialize the entire instruction or data caches by stepping through all valid indices. Doing so requires that the TagLo and TagHi registers associated with the cache be initialized first.</td><td>Yes</td></tr>
<tr><td>2#011</td><td>All</td><td>Reserved</td><td>Unspecified</td><td>Executed as a no-op.</td><td>No</td></tr>
</table>

Table 11-13 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST,SPR] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	Implemented?
2#100	I, D	Hit Invalidate	Address	If the cache block contains the specified address, set the state of the cache block to invalid. This encoding may be used by software to invalidate a range of addresses from the instruction cache by stepping through the address range by the line size of the cache.	Yes
	S, T	Reserved	Address		No
2#101	I	Fill	Address	Fill the cache from the specified address. The cache line is refetched even if it is already in the cache.	Yes
	D	Hit Writeback Invalidate	Address	If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After that operation is completed, set the state of the cache block to invalid. If the block is valid but not dirty, set the state of the block to invalid.This encoding may be used by software to invalidate a range of addresses from the data cache by stepping through the address range by the line size of the cache.	Yes
	S, T	Reserved	Address		No
2#110	D	Hit Writeback	Address	If the cache block contains the specified address and it is valid and dirty, write the contents back to memory. After the operation is completed, leave the state of the line valid, but clear the dirty state.	Yes
	S, T	Reserved	Address		No

Table 11-13 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST,SPR] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	Implemented?
2#111	I, D	Fetch and Lock	Address	If the cache does not contain the specified address, fill it from memory, performing a writeback if required, and set the state to valid and locked. If the cache already contains the specified address, set the state to locked. The way selected on fill from memory is the least recently used. The lock state is cleared by executing an Index Invalidate, Index Writeback Invalidate, Hit Invalidate, or Hit Writeback Invalidate operation to the locked line, or via an Index Store Tag operation with the lock bit reset in the *TagLo* register.	Yes

Table 11-14 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST] Set. ErrCtl[SPR] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	Implemented?
2#001	I, D	Index Load WS	Index	Read the WS RAM at the specified index into the *TagLo* Coprocessor 0 register.	Yes
2#010	I, D	Index Store WS	Index	Update the WS RAM at the specified index from the *TagLo* Coprocessor 0 register.	Yes
2#011	I, D	Index Store Data	Index	Write the *DataLo* Coprocessor 0 register contents at the way and byte index specified.	Yes
All Others	All			All of the other codes behave the same as when ErrCtl[WST] is cleared.	

Table 11-15 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[SPR] Set

Code	Caches	Name	Effective Address Operand Type	Operation	Implemented?
2#001	I, D	Index Load Tag	Index	Read the SPRAM tag at the specified index into the *TagLo* Coprocessor 0 register. Also read the data corresponding to the byte index into the *DataLo* register	Yes
2#010	I, D	Index Store Tag	Index	Update the SPRAM tag at the specified index from the *TagLo* Coprocessor 0 register.	Yes
2#011	I, D	Index Store Data	Index	Write the *DataLo* Coprocessor 0 register contents into the SPRAM at the word index specified.	Yes
All Others	All			All of the other codes behave the same as when ErrCtl[SPR] is cleared.	

Restrictions:

The operation of this instruction is **UNDEFINED** for any operation/cache combination that is not implemented.

The operation of this instruction is **UNDEFINED** if the operaation requires an address, and that address is uncacheable.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, uncached) ← AddressTranslation(vAddr, DataReadReference)
CacheOp(op, vAddr, pAddr)
```

Exceptions:

TLB Refill Exception.

TLB Invalid Exception

Coprocessor Unusable Exception

Address Error Exception

Bus Error Exception

31–26	25–21	20–16	15–0
LL 110000	base	rt	offset
6	5	5	16

Format: `LL rt, offset(base)` **MIPS32**

Purpose:

To load a word from memory for an atomic read-modify-write

Description: `rt ← memory[base+offset]`

The LL and SC instructions provide the primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and written into GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

This begins a RMW sequence on the current processor. There can be only one active RMW sequence per processor. When an LL is executed it starts an active RMW sequence replacing any other sequence that was active. The RMW sequence is completed by a subsequent SC instruction that either completes the RMW sequence atomically and succeeds, or does not and fails.

Executing LL on one processor does not cause an action that, by itself, causes an SC for the same block to fail on another processor.

An execution of LL does not have to be followed by execution of SC; a program is free to abandon the RMW sequence without attempting a write.

Restrictions:

The addressed location must be synchronizable by all processors and I/O devices sharing the location; if it is not, the result in **UNPREDICTABLE**. Which storage is synchronizable is a function of both CPU and system implementations. See the documentation of the SC instruction for the formal definition.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0² then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
LLbit  ← 1
```

Exceptions:

TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Watch

Programming Notes:

There is no Load Linked Word Unsigned operation corresponding to Load Word Unsigned.

Prefetch **PREF**

31–26	25–21	20–16	15–0
PREF 110011	base	hint	offset
6	5	5	16

Format: `PREF hint,offset(base)` **MIPS32**

Purpose:

To move data between memory and cache.

Description: `prefetch_memory(base+offset)`

PREF adds the 16-bit signed *offset* to the contents of GPR *base* to form an effective byte address. The *hint* field supplies information about the way that the data is expected to be used.

PREF is an advisory instruction that may change the performance of the program. However, for all *hint* values except for PrepareForStore, and all effective addresses, it neither changes the architecturally visible state nor does it alter the meaning of the program.

PREF does not cause addressing-related exceptions. If the address specified would cause an addressing exception, the exception condition is ignored and no data movement occurs.However even if no data is prefetched, some action that is not architecturally visible, such as writeback of a dirty cache line, can take place.

PREF never generates a memory operation for a location with an *uncached* memory access type.

If PREF results in a memory operation, the memory access type used for the operation is determined by the memory access type of the effective address, just as it would be if the memory operation had been caused by a load or store to the effective address.

The *hint* field supplies information about the way the data is expected to be used. With the exception of PrepareForStore, a *hint* value cannot cause an action to modify architecturally visible state.

Any of the following conditions causes the core to treat a PREF instruction as a NOP.

- A reserved *hint* value is used
- The address has a translation error
- The address maps to an uncacheable page

In all other cases, except when *hint* equals 25, execution of the PREF instruction initiates an external bus read transaction. PREF is a non-blocking operation and does not cause the pipeline to stall while waiting for the data to be returned.

Table 11-16 Values of the *hint* Field for the PREF Instruction

Value	Name	Data Use and Desired Prefetch Action
0	load	Use: Prefetched data is expected to be read (not modified). Action: Fetch data as if for a load.
1	store	Use: Prefetched data is expected to be stored or modified. Action: Fetch data as if for a store.
2-3	Reserved	Reserved - treated as a NOP.
4	load_streamed	Use: Prefetched data is expected to be read (not modified) but not reused extensively; it “streams” through cache. Action: Fetch data as if for a load and place it in the cache so that it does not displace data prefetched as “retained.”
5	store_streamed	Use: Prefetched data is expected to be stored or modified but not reused extensively; it “streams” through cache. Action: Fetch data as if for a store and place it in the cache so that it does not displace data prefetched as “retained.”
6	load_retained	Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a load and place it in the cache so that it is not displaced by data prefetched as “streamed.”
7	store_retained	Use: Prefetched data is expected to be stored or modified and reused extensively; it should be “retained” in the cache. Action: Fetch data as if for a store and place it in the cache so that it is not displaced by data prefetched as “streamed.”

Table 11-16 Values of the *hint* Field for the PREF Instruction

8-24	Reserved	Reserved - treated as a NOP.
25	writeback_invalidate (also known as "nudge")	Use: Data is no longer expected to be used. Action: Schedule a writeback of any dirty data. The cache line is marked as invalid upon completion of the writeback. If cache line is clean or locked, no action is taken.
26-29	Reserved	Reserved - treated as a NOP.
30	PrepareForStore	Use: Prepare the cache for writing an entire line, without the overhead involved in filling the line from memory. Reserved - treated as a NOP.
31	Reserved	Reserved - treated as a NOP.

Restrictions:

None

Operation:

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

Programming Notes:

Prefetch cannot prefetch data from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. It does not cause an exception to prefetch using an address pointer value before the validity of a pointer is determined.

Prefetch operations have no effect on cache lines that were previously locked with the CACHE instruction.

Store Conditional Word **SC**

31–26	25–21	20–16	15–0
SC 111000	base	rt	offset
6	5	5	16

Format: `SC rt, offset(base)` **MIPS32**

Purpose:

To store a word to memory to complete an atomic read-modify-write

Description: `if atomic_update then memory[base+offset] ← rt, rt ← 1 else rt ← 0`

The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The 32-bit word in GPR *rt* is conditionally stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The SC completes the RMW sequence begun by the preceding LL instruction executed on the processor. To complete the RMW sequence atomically, the following occur:

- The 32-bit word of GPR *rt* is stored into memory at the location specified by the aligned effective address.
- A 1, indicating success, is written into GPR *rt*.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR *rt*.

If the following event occurs between the execution of LL and SC, the SC fails:

- An ERET instruction is executed.

If either of the following events occurs between the execution of LL and SC, the SC may succeed or it may fail; the success or failure is not predictable. Portable programs should not cause one of these events.

- A memory access instruction (load, store, or prefetch) is executed on the processor executing the LL/SC.
- The instructions executed starting with the LL and ending with the SC do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

The following conditions must be true or the result of the SC is **UNPREDICTABLE**:

- Execution of SC must have been preceded by execution of an LL instruction.
- An RMW sequence executed without intervening events that would cause the SC to fail must use the same address in the LL and SC. The address is the same if the virtual address, physical address, and cache-coherence algorithm are identical.

Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
dataword← GPR[rt]
if LLbit then
   StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
endif
GPR[rt]← 0^31 || LLbit
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

Programming Notes:

LL and SC are used to atomically update memory locations, as shown below.

```
L1:
    LL      T1, (T0)  # load counter
    ADDI    T2, T1, 1 # increment
    SC      T2, (T0)  # try to store, checking for atomicity
    BEQ     T2, 0, L1 # if not atomic (0), try again
    NOP               # branch-delay slot
```

Exceptions between the LL and SC cause SC to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LL and SC function on a single processor for *cached noncoherent* memory so that parallel programs can be run on uniprocessor systems that do not support *cached coherent* memory access types.

Synchronize Shared Memory — SYNC

31–26	25–11	10–6	5–0
SPECIAL 000000	0 00 0000 0000 0000 0	stype	SYNC 001111
6	15	5	6

Format: `SYNC (stype = 0 implied)` **MIPS32**

Purpose:

To order loads and stores.

Description:

Simple Description:

- SYNC affects only *uncached* and *cached coherent* loads and stores. The loads and stores that occur before the SYNC must be completed before the loads and stores after the SYNC are allowed to start.
- Loads are completed when the destination register is written. Stores are completed when the stored value is visible to every other processor in the system.
- SYNC is required, potentially in conjunction with SSNOP, to guarantee that memory reference results are visible across operating mode changes. For example, a SYNC is required on entry to and exit from Debug Mode to guarantee that memory affects are handled correctly.

Detailed Description:

- SYNC does not guarantee the order in which instruction fetches are performed. The *stype* values 1-31 are reserved for future extensions to the architecture. A value of zero will always be defined such that it performs all defined synchronization operations. Non-zero values may be defined to remove some synchronization operations. As such, software should never use a non-zero value of the *stype* field, as this may inadvertently cause future failures if non-zero values remove synchronization operations.
- The SYNC instruction stalls until all loads, stores, refills are completed and all write buffers are empty.

Restrictions:

The effect of SYNC on the global order of loads and stores for memory access types other than *uncached* and *cached coherent* is **UNPREDICTABLE**.

Operation:

```
SyncOperation(stype)
```

Exceptions:

None

Read Indexed TLB Entry — TLBR

31–26	25	24–6	5–0
COP0	CO	0	TLBR
010000	1	000 0000 0000 0000 0000	000001
6	1	19	6

Format: `TLBR` **MIPS32**

Purpose:

To read an entry from the TLB.

Description:

The *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers are loaded with the contents of the TLB entry pointed to by the Index register. Note that the value written to the *EntryHi*, *EntryLo0*, and *EntryLo1* registers may be different from that originally written to the TLB via these registers in that:

- The value returned in the G bit in both the *EntryLo0* and *EntryLo1* registers comes from the single G bit in the TLB entry. Recall that this bit was set from the logical AND of the two G bits in *EntryLo0* and *EntryLo1* when the TLB was written.

Restrictions:

The operation is **UNDEFINED** if the contents of the Index register are greater than or equal to the number of TLB entries in the processor.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
i ← Index
if i > (TLBEntries - 1) then
    UNDEFINED
endif
PageMask_Mask ← TLB[i]_Mask
EntryHi ←
            TLB[i]_VPN2 ||
            0^5 || TLB[i]_ASID
EntryLo1 ← 0^2 ||
            TLB[i]_PFN1 ||
            TLB[i]_C1 || TLB[i]_D1 || TLB[i]_V1 || TLB[i]_G
EntryLo0 ← 0^2 ||
            TLB[i]_PFN0 ||
            TLB[i]_C0 || TLB[i]_D0 || TLB[i]_V0 || TLB[i]_G
```

Exceptions:

Coprocessor Unusable

Write Indexed TLB Entry — TLBWI

31–26	25	24–6	5–0
COP0	CO	0	TLBWI
010000	1	000 0000 0000 0000 0000	000010
6	1	19	6

Format: `TLBWI` **MIPS32**

Purpose:

To write a TLB entry indexed by the *Index* register.

Description:

The TLB entry pointed to by the Index register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in that:

- The single G bit in the TLB entry is set from the logical AND of the G bits in the *EntryLo0* and *EntryLo1* registers.

Restrictions:

The operation is **UNDEFINED** if the contents of the Index register are greater than or equal to the number of TLB entries in the processor.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

$i \leftarrow$ Index
$TLB[i]_{Mask} \leftarrow PageMask_{Mask}$
$TLB[i]_{VPN2} \leftarrow EntryHi_{VPN2}$
$TLB[i]_{ASID} \leftarrow EntryHi_{ASID}$
$TLB[i]_{G} \leftarrow EntryLo1_{G}$ and $EntryLo0_{G}$
$TLB[i]_{PFN1} \leftarrow EntryLo1_{PFN}$
$TLB[i]_{C1} \leftarrow EntryLo1_{C}$
$TLB[i]_{D1} \leftarrow EntryLo1_{D}$
$TLB[i]_{V1} \leftarrow EntryLo1_{V}$
$TLB[i]_{PFN0} \leftarrow EntryLo0_{PFN}$
$TLB[i]_{C0} \leftarrow EntryLo0_{C}$
$TLB[i]_{D0} \leftarrow EntryLo0_{D}$
$TLB[i]_{V0} \leftarrow EntryLo0_{V}$

Exceptions:

Coprocessor Unusable

Write Random TLB Entry — TLBWR

31–26	25	24–6	5–0
COP0	CO	0	TLBWR
010000	1	000 0000 0000 0000 0000	000110
6	1	19	6

Format: `TLBWR` **MIPS32**

Purpose:

To write a TLB entry indexed by the *Random* register.

Description:

The TLB entry pointed to by the *Random* register is written from the contents of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers. The information written to the TLB entry may be different from that in the *EntryHi*, *EntryLo0*, and *EntryLo1* registers, in that:

- The single G bit in the TLB entry is set from the logical AND of the G bits in the *EntryLo0* and *EntryLo1* registers.

Restrictions:

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

$i \leftarrow \text{Random}$
$TLB[i]_{Mask} \leftarrow PageMask_{Mask}$
$TLB[i]_{VPN2} \leftarrow EntryHi_{VPN2}$
$TLB[i]_{ASID} \leftarrow EntryHi_{ASID}$
$TLB[i]_{G} \leftarrow EntryLo1_{G}$ and $EntryLo0_{G}$
$TLB[i]_{PFN1} \leftarrow EntryLo1_{PFN}$
$TLB[i]_{C1} \leftarrow EntryLo1_{C}$
$TLB[i]_{D1} \leftarrow EntryLo1_{D}$
$TLB[i]_{V1} \leftarrow EntryLo1_{V}$
$TLB[i]_{PFN0} \leftarrow EntryLo0_{PFN}$
$TLB[i]_{C0} \leftarrow EntryLo0_{C}$
$TLB[i]_{D0} \leftarrow EntryLo0_{D}$
$TLB[i]_{V0} \leftarrow EntryLo0_{V}$

Exceptions:

Coprocessor Unusable

Enter Standby Mode **WAIT**

31 26	25	24 6	5 0
COP0 010000	CO 1	Implementation-Dependent Code	WAIT 100000
6	1	19	6

Format: `WAIT` **MIPS32**

Purpose:

Wait for Event

Description:

The WAIT instruction forces the core into low power mode. The pipeline is stalled and when all external requests are completed, the processor's main clock is stopped. The processor will restart when reset (SI_Reset or SI_ColdReset) is signaled, or a non-masked interrupt is taken (SI_NMI, SI_Int, or EJ_DINT). Note that the 4KE core does not use the code field in this instruction.

If the pipeline restarts as the result of an enabled interrupt, that interrupt is taken between the WAIT instruction and the following instruction (EPC for the interrupt points at the instruction following the WAIT instruction).

Restrictions:

The operation of the processor is **UNDEFINED** if a WAIT instruction is placed in the delay slot of a branch or a jump.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
I:  Enter  lower power mode
I+1:/* Potential interrupt taken here */
```

Exceptions:

Coprocessor Unusable Exception

MIPS16 Application-Specific Extension to the MIPS32 Instruction Set

This chapter describes the MIPS16 ASE as implemented in the 4KE core. Refer to Volume IV-a of the MIPS32 Architecture Reference Manual for a general description of the MIPS16 ASE as well as instruction descriptions.

This chapter covers the following topics:

12.1 Instruction Bit Encoding

Table 12-2 through Table 12-9 describe the encoding used for the MIPS16 ASE. Table 12-1 describes the meaning of the symbols used in the tables.

Table 12-1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction cause a Reserved Instruction Exception.
δ	(Also *italic* field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.
β	Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level. Executing such an instruction cause a Reserved Instruction Exception.
θ	Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, the partner must notify MIPS Technologies, Inc. when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (*SPECIAL2* encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
σ	Field codes marked with this symbol represent an EJTAG support instruction and implementation of this encoding is optional for each implementation. If the encoding is not implemented, executing such an instruction must cause a Reserved Instruction Exception. If the encoding is implemented, it must match the instruction encoding as shown in the table.
ε	Operation or field codes marked with this symbol are reserved for MIPS Application Specific Extensions. If the ASE is not implemented, executing such an instruction must cause a Reserved Instruction Exception.
ϕ	Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS64 ISA. Software should avoid using these operation or field codes.

Table 12-2 MIPS16 Encoding of the Opcode Field

opcode	bits 13..11								
		0	1	2	3	4	5	6	7
bits 15..14		000	001	010	011	100	101	110	111
0	00	ADDIUSP[1]	ADDIUPC[2]	B	*JAL(X)* δ	BEQZ	BNEZ	*SHIFT* δ	β
1	01	*RRI-A* δ	ADDIU8[3]	SLTI	SLTIU	*I8* δ	LI	CMPI	β
2	10	LB	LH	LWSP[4]	LW	LBU	LHU	LWPC[5]	β
3	11	SB	SH	SWSP[6]	SW	*RRR* δ	*RR* δ	*EXTEND* δ	β

1. The ADDIUSP opcode is used by the ADDIU rx, sp, immediate instruction
2. The ADDIUPC opcode is used by the ADDIU rx, pc, immediate instruction
3. The ADDIU8 opcode is used by the ADDIU rx, immediate instruction
4. The LWSP opcode is used by the LW rx, offset(sp) instruction
5. The LWPC opcode is used by the LW rx, offset(pc) instruction
6. The SWSP opcode is used by the SW rx, offset(sp) instruction

Table 12-3 MIPS16 JAL(X) Encoding of the x Field

x	bit 26	
	0	1
	JAL	JALX

Table 12-4 MIPS16 SHIFT Encoding of the f Field

f	bits 1..0			
	0	1	2	3
	00	01	10	11
	SLL	β	SRL	SRA

Table 12-5 MIPS16 RRI-A Encoding of the f Field

f	bit 4	
	0	1
	ADDIU[1]	β

1. The ADDIU function is used by the ADDIU ry, rx, immediate instruction

Table 12-6 MIPS16 I8 Encoding of the funct Field

funct	bits 10..8							
	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
	BTEQZ	BTNEZ	SWRASP[1]	ADJSP[2]	*SVRS* δ	MOV32R[3]	*	MOVR32[4]

1. The SWRASP function is used by the SW ra, offset(sp) instruction
2. The ADJSP function is used by the ADDIU sp, immediate instruction
3. The MOV32R function is used by the MOVE r32, rz instruction
4. The MOVR32 function is used by the MOVE ry, r32 instruction

Table 12-7 MIPS16 RRR Encoding of the f Field

f	bits 1..0			
	0	1	2	3
	00	01	10	11
	β	ADDU	β	SUBU

Table 12-8 MIPS16 RR Encoding of the Funct Field

funct		bits 2..0							
		0	1	2	3	4	5	6	7
bits 4..3		000	001	010	011	100	101	110	111
0	00	*J(AL)R(C)* δ	SDBBP	SLT	SLTU	SLLV	BREAK	SRLV	SRAV
1	01	β	*	CMP	NEG	AND	OR	XOR	NOT
2	10	MFHI	*CNVT* δ	MFLO	β	β	*	β	β
3	11	MULT	MULTU	DIV	DIVU	β	β	β	β

Table 12-9 MIPS16 I8 Encoding of the s Field when funct=SVRS

s	bit 7	
	0	1
	RESTORE	SAVE

Table 12-10 MIPS16 RR Encoding of the ry Field when funct=J(AL)R(C)

ry	bits 7..5							
	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
	JR rx	JR ra	JALR	*	JRC rx	JRC ra	JALRC	*

Table 12-11 MIPS16 RR Encoding of the ry Field when funct=CNVT

ry	bits 7..5							
	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
	ZEB	ZEH	β	*	SEB	SEH	β	*

12.2 Instruction Listing

Table 12-12 through 12-19 list the MIPS16 instruction set.

Table 12-12 MIPS16 Load and Store Instructions

Mnemonic	Instruction	Extensible Instruction
LB	Load Byte	Yes
LBU	Load Byte Unsigned	Yes
LH	Load Halfword	Yes

Table 12-12 MIPS16 Load and Store Instructions

Mnemonic	Instruction	Extensible Instruction
LHU	Load Halfword Unsigned	Yes
LW	Load Word	Yes
SB	Store Byte	Yes
SH	Store Halfword	Yes
SW	Store Word	Yes

Table 12-13 MIPS16 Save and Restore Instructions

Mnemonic	Instruction	Extensible Instruction
RESTORE	Restore Registers and Deallocate Stack Frame	Yes
SAVE	Save Registers and Setup Stack Frame	Yes

Table 12-14 MIPS16 ALU Immediate Instructions

Mnemonic	Instruction	Extensible Instruction
ADDIU	Add Immediate Unsigned	Yes
CMPI	Compare Immediate	Yes
LI	Load Immediate	Yes
SLTI	Set on Less Than Immediate	Yes
SLTIU	Set on Less Than Immediate Unsigned	Yes

Table 12-15 MIPS16 Arithmetic Two or Three Operand Register Instructions

Mnemonic	Instruction	Extensible Instruction
ADDU	Add Unsigned	No
AND	AND	No
CMP	Compare	No
MOVE	Move	No
NEG	Negate	No
NOT	Not	No
OR	OR	No
SEB	Sign-Extend Byte	No
SEH	Sign-Extend Halfword	No

Table 12-15 MIPS16 Arithmetic Two or Three Operand Register Instructions

Mnemonic	Instruction	Extensible Instruction
SLT	Set on Less Than	No
SLTU	Set on Less Than Unsigned	No
SUBU	Subtract Unsigned	No
XOR	Exclusive OR	No
ZEB	Zero-Extend Byte	No
ZEH	Zero-Extend Halfword	No

Table 12-16 MIPS16 Special Instructions

Mnemonic	Instruction	Extensible Instruction
BREAK	Breakpoint	No
SDBBP	Software Debug Breakpoint	No
EXTEND	Extend	No

Table 12-17 MIPS16 Multiply and Divide Instructions

Mnemonic	Instruction	Extensible Instruction
DIV	Divide	No
DIVU	Divide Unsigned	No
MFHI	Move From HI	No
MFLO	Move From LO	No
MULT	Multiply	No
MULTU	Multiply Unsigned	No

Table 12-18 MIPS16 Jump and Branch Instructions

Mnemonic	Instruction	Extensible Instruction
B	Branch Unconditional	Yes
BEQZ	Branch on Equal to Zero	Yes
BNEZ	Branch on Not Equal to Zero	Yes
BTEQZ	Branch on T Equal to Zero	Yes
BTNEZ	Branch on T Not Equal to Zero	Yes
JAL	Jump and Link	No

Table 12-18 MIPS16 Jump and Branch Instructions

Mnemonic	Instruction	Extensible Instruction
JALR	Jump and Link Register	No
JALRC	Jump and Link Register Compact	No
JALX	Jump and Link Exchange	No
JR	Jump Register	No
JRC	Jump Register Compact	No

Table 12-19 MIPS16 Shift Instructions

Mnemonic	Instruction	Extensible Instruction
SRA	Shift Right Arithmetic	Yes
SRAV	Shift Right Arithmetic Variable	No
SLL	Shift Left Logical	Yes
SLLV	Shift Left Logical Variable	No
SRL	Shift Right Logical	Yes
SRLV	Shift Right Logical Variable	No

Revision History

Table A-1 Revision History

Revision	Date	Description
0.90	November 13, 2000	First preliminary version
0.91	November 17, 2000	Changes for this revision: • Added LWC2 and SWC2 to opcode map Table 11-1 on page 237 • Updated TagLo CP0 register format for new handling of LRU bits • Added ErrCtl CP0 register • Added more details to WS description in cache chapter • Added description of how to test the cache arrays in software.
0.93	February 16, 2001	• Instruction and data micro TLBs in the 4KEc are now 4 entries (previously 3). • Added support for 64KB maximum cache sizes. • Added support for write-through with write-allocate cache policy. • Enhanced description of PrID revision field. • Added discussion about virtual aliasing in the caches.
01.00	March 27, 2001	• Removed extraneous reference to “Supervisor mode” in Table 11-1 on page 191, since Supervisor mode is not supported. • Standardized links to major sections in each chapter. • Added SimpleBE & UDI config bits. Cleaned up description of Config registers. • Added note about ASID field in *EntryHi* not being updated on an exception. • Updated descriptions of CACHE, PREF, and SYNC to include processor specific information.
01.01	April 2, 2001	• Added note that it is invalid to have all ways locked in the data cache (no longer invalid, superseded by revision 1.07).
01.02	May 16, 2001	• Added WST=1 table to CACHE instruction description
01.03	June 12, 2001	• Minor changes in the instruction decode tables. • Added details on new mechanism for CACHE access to ScratchPad RAMs. • Removed support for MIPS16 ASMACRO. • Modified text and reset state for CU2 bit in *Status* register, and updated text on C2 bit in *Config1*.

Table A-1 Revision History (Continued)

Revision	Date	Description
01.04	July 16, 2001	• Added MIPS16 bit in EJTAG Implementation register. • Added missing footnote in Table 2-6 on page 29. • Fixed typo in LSNM field description in Table 5-34 on page 133. • Correct name of ASIDsup field in description of IBS (Table 9-7 on page 174) and DBS (Table 9-13 on page 180) registers. • Correct name of ASIDuse field in description of IBCn (Table 9-11 on page 178) and DBCn (Table 9-17 on page 184) registers. • Added definitions of UNDEFINED and UNPREDICTABLE. • Added definitions of precise and imprecise exception (Chapter 4, “Exceptions and Interrupts,” on page 53).
01.05	August 30, 2001	• Removed common instruction descriptions. Instructions with processor specific behavior are included here, refer to architecture documents for others. • Noted that interrupts are not prioritized by the HW. Changed example for long interrupt latency instruction from SYNC to uncached load. • Added CP0 PDtrace register in Chapter 5, “CP0 Registers.” • Added EJTAG Trace sections in Chapter 9, “EJTAG Debug Support.” • Added FastData description to Section 9.3, "Test Access Port (TAP)" on page 187. • Changed EJTAGver field from 1 -> 2 (version 2.5 to 2.6), in Section 9.4.2.3, "Implementation Register" on page 195.
01.06	October 4, 2001	• Added SDDBP MIPS16 instruction to Table 12-16 “Special Instructions”. • Marked unused J(AL)R(C) encodings as reserved. • Removed obsolete references to 2-bit ISA mode field. • Corrected the heading format in Section 10.2.1, "Scheduling a Load Delay Slot" on page 232. • Changed confidentiality level to “commercial”.
01.07	December 5, 2001	• Clarified handling of all locked cache ways.
01.08	January 30, 2002	• EJTAG Version field in Debug register is set to 010 • Added description for constant fields in Debug register: NoDCR, NoSSt, MCheckP, CacheEP, DDBSImpr, DDBLImpr •
02.00	November 8, 2002	• Major update for addition of MIPS32 Release 2 features. • Added support for 64MB and 256MB pages in TLB (4KEc core only). • Wrong bit of MM field in *Config* register was being used. Describe as 2b field now. • Address region for DSEG was wrong in figure in memory management chapter.